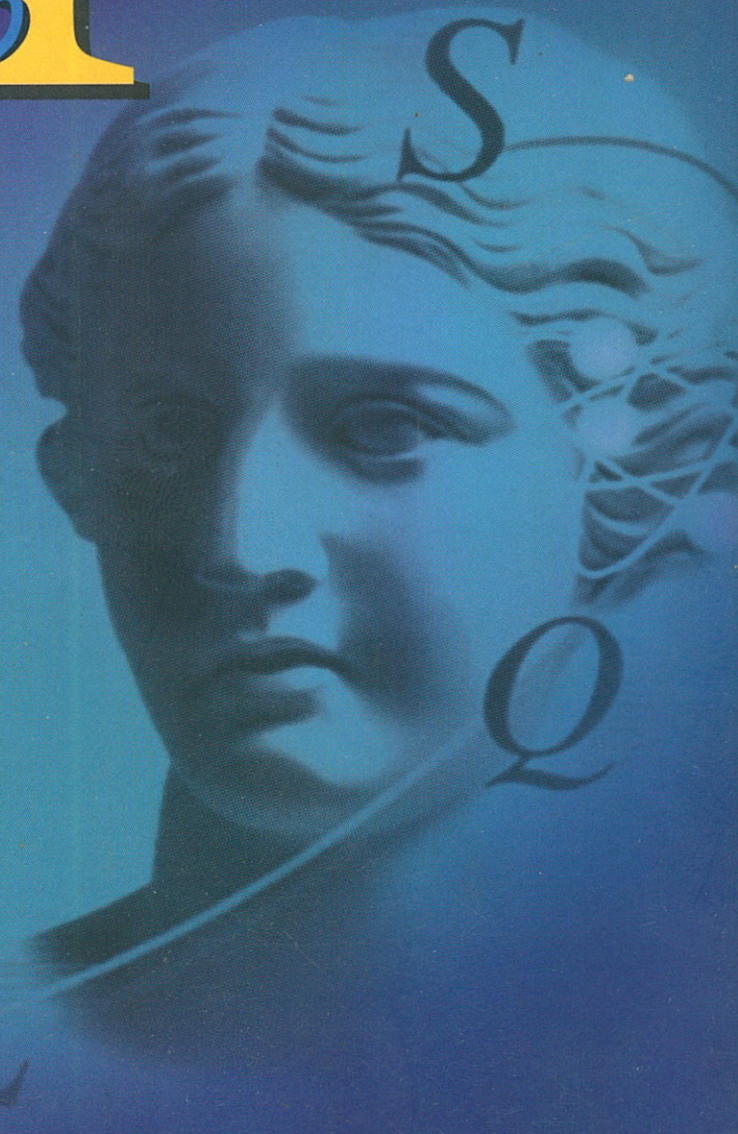
BAGA EDIT

Delphi

másképp



Alapok
Adatbázisok
Kliens/szerver
Multi-tier
OLE, DDE
Multi-thread
... és egyebek

Baga Edit

Delphi másképp...

GÁBOR DÉNES
FŐISKOLA
részére

© Baga Edit, 1998

Második kiadás, 1999 március

Minden jog fenntartva. A szerző előzetes írásbeli engedélye nélkül a könyvet semmilyen formában nem szabad reprodukálni.

Szerkesztette: Baga Edit

Tanácsadó és lektor: Baga Zoltán

Borítóterv: Nagy Ágnes

ISBN 963 03 5066 1

Akadémiai Nyomda, Martonvásár
Felelős vezető: Reisenleitner Lajos

Előszó

Rohanó világban élünk, ahol a legnagyobb probléma az állandó időhiány. Ez a számítástechnikában is így van. A hardver eszközök rohamosan fejlődnek, az emberek szoftverrel szemben támasztott igényeik egyre nagyobbak. A programozóknak ugyanannyi vagy talán kevesebb idő alatt sokkal szemléletesebb, barátságosabb és természetesen az elvárásoknak megfelelően működő, megbízható alkalmazásokat kell gyártaniuk. Talán emiatt terjedt el, és terjed napjainkban is egyre több területen az objektumorientált szemlélet. Egyre nagyobb teret hódítanak a vizuális negyedik generációs fejlesztőeszközök is, melyekben adatbázis-specifikáló, képernyőtervező és jelentéstervező eszközök segítségével lényegesen kevesebb kódolással készíthetjük el alkalmazásainkat. A Delphi egy olyan negyedik generációs fejlesztőeszköz, melynek nyelve – az Object Pascal – egy igazi objektumorientált nyelv. Rengeteg (Delphi 3-ban kb. 150) kész komponens áll rendelkezésünkre, ezek segítségével könnyűszerrel létrehozhatunk tetszetős űrlapokat, bonyolult lebomló menüket, adatkezelő rendszereket és akár még internetes alkalmazásokat is. És ha mindez nem lenne elég, akkor saját komponenseket is fejleszthetünk, sőt válogathatunk a megannyi, világhálón található, mások által fejlesztettek közül is.

Azt szokták mondani, hogy a 4GL-ekben pillanatokon belül pár ide-oda kattintással fel lehet építeni egy „hét nyelven beszélő" alkalmazást. Ezt az állítást én nem kívánom megcáfolni, csupán kiegészíteni: való igaz, hogy pár mozdulattal annyit valósítunk meg, mint más hagyományos eszközzel több napi munka árán, de azt sem szabad szem elől tévesztenünk, hogy e mozdulatoknak jól irányítottaknak kell lenniük, azaz nagyon jól ismernünk kell a rendelkezésünkre álló eszközt és ennek rengeteg lehetőségét. Folytatva a gondolatmenetet, csak akkor leszünk képesek jó, hatékony alkalmazást „összedobni", ha mindezeken felül egy kicsit belátunk a színfalak mögé is, vagyis az alkalmazást ésszel, céltudatosan „kattintgatjuk össze". Véleményem szerint ezen ismeretek hiányában megmaradnánk „egy okos eszköz buta felhasználóinak".
Mindnyájunkat feldob az első 4GL rendszerben fejlesztett alkalmazásunk. De álljunk meg egy szóra! Mitől fut ez a program? Erre a kérdésre és sok másra is választ igyekeztem adni könyvemben. Lépésről-lépésre haladva, elméleti fogalmak, majd konkrét megoldott feladatok segítségével ismerkedhetünk meg a Delphi lehetőségeivel.

A könyvet minden érdeklődő – Delphi-vel alig most ismerkedő, vagy már „veteránnak" számító – Olvasónak ajánlom. Szándékaim szerint a könyv mindenki számára tartalmaz érdekes részeket. Az első részt (1.-6. fejezet) a Delphivel most ismerkedő Olvasóimnak javaslom. A második egység (7.-14. fejezet) a Delphiben történő adatbázis-kezelést mutatja

be. A könyv harmadik részében (15.-20. fejezet) szerteágazó, de ugyanakkor érdekes és „divatos" témákkal foglalkozunk, mint például az OLE, DDE, multi-tier, multi-thread, súgó írása, komponensek fejlesztése... Minden egyes rész elején található egy bevezető az illető rész elsajátításához szükséges előképzettséggel, valamint fejezeteinek rövid ismertetésével.

A fejlesztőeszközök használatát nem lehet csak könyvből, elméletben megtanulni. Emiatt igyekeztem könyvemben minél több megoldott feladatot bemutatni (közel 60 alkalmazást), és a használt technikákat, valamint a teljes gondolatmenetet elmagyarázni. Ez azonban csak akkor lesz igazán hatékony, ha velem párhuzamosan Ön is megpróbálja megoldani a kitűzött feladatokat. Ehhez nyújthat segítséget a könyv mellékletének szánt megoldott feladatok gyűjteménye, mely az Interneten az alábbi két címről tölthető le:

www.borland.hu: könyvek oldal
ftp.gdf.hu/public/prog/delphi.zip

Persze felvetődhet a kedves Olvasóban a kérdés, hogy ezek a feladatok a Delphi mely verziójában készültek? A könyvben a 3-as verziót használtam, de mivel kishazánkban még a régebbi – a 16 bites Delphi 1 és a már 32 bites Delphi 2 – változatok is széleskörűen elterjedtek, a különböző technikák bemutatásánál igyekeztem ezekre is kitérni.
Az első kiadás óta megjelent a Delphi 4-es verziója. A könyvben leírtak erre a környezetre is vonatkoznak, a feladatok Delphi 4-esben is futtathatók.

Ezúton szeretnék köszönetet mondani mindazoknak, akik e könyv elkészítésében segítségemre voltak. Első sorban meg szeretném köszönni Angster Erzsébetnek az évek során nyújtott támogatását, értékes tanácsait, folyamatos bátorítását. Hálás vagyok Naszádi Gábornak, aki a Delphi világában tett első lépteimet kalauzolta. Köszönöm Zsembery Ágostonnak a könyvhöz fűzött értékes hozzászólásait, építő kritikáját. Továbbá köszönetemet fejezem ki Andor Gergőnek, Apáti Jánosnak, Siposné Németh Mariannának és Tuba Lászlónak, akik különös gonddal olvasták a készülőfélben levő könyvet, segítséget nyújtva a hibák kijavításában.
Ugyancsak itt szeretném kifejezni hálámat Márton Attilánénak, Szabó Zoltánnénak, Nagy Sándornak, Jánosik Gáspárnak, Nagy Zoltánnak és családjaiknak az évekkel ezelőtt nyújtott segítségükért, mely későbbi szakmai pályafutásomat jelentősen befolyásolta.

Kedves Olvasó! Szeretném, ha a könyvvel kapcsolatos észrevételeit, tapasztalatait – legyenek azok jók vagy rosszak – velem is megosztaná. E-mail címem a következő:

bagae@hotmail.com

Tartalomjegyzék

I. RÉSZ: ALAPFOGALMAK

1. Windows bevezetés 3

1.1 Általános tudnivalók 3
1.2 Windows eseménykezelés, üzenetvezérlés 9
1.2.1 Az esemény és az üzenet fogalma 9
1.2.2 Az üzenetek típusai 11
1.2.3 Üzenetek szétszórása és feldolgozása 12
1.2.4 Multitasking a 16 és 32 bites Windows verziókban 15
1.2.5 Nem sorolt üzenetek 16

2. Delphi bevezetés 17

2.1 A Delphi alkalmazások felépítése 17
2.1.1 A projektállomány szerkezete (*.DPR) 20
2.1.2 Az űrlapállomány szerkezete (*.DFM) 22
2.1.3 Az űrlaphoz tartozó egység (*.PAS) 24
2.2 Egy egyszerű Delphi alkalmazás elkészítése 28

3. A Turbo Pascaltól az Object Pascalig 31

3.1 A különbségek és újdonságok rövid áttekintése 31
3.2 Új, hasznos rutinok 32
3.3 Az Object Pascal osztálymodell 33
3.3.1 Az osztály deklarációja 33
3.3.2 Mezőlista 36
3.3.3 Metóduslista 36
3.3.4 Az adathozzáférés korlátozása 41
3.3.5 Jellemzők 42
3.3.6 Osztályoperátorok 46
3.4 Összetett típusú függvényértékek 48
3.5 Kivételek kezelése *(Exception handling)* 48
3.5.1 Védelem a futás-idejű hibák ellen (*Try...Except*) 50
3.5.2 Erőforrások biztonságos használata (Try...Finally) 53
3.5.3 Saját kivételek létrehozása 54
3.5.4 Kivételek ismételt előidézése 55
3.6 Object Pascal karakterláncok 56

4. Delphi standard komponensek 59

4.1 TComponent 60
4.2 TControl 62
4.3 TLabel (címke) 65
4.4 TWinControl 65

4.5 TEdit (szerkesztődoboz) 66
4.6 TMemo (többsoros szerkesztődoboz), TRichEdit 67
4.7 TButton (gomb), TBitBtn (Additional paletta) 68
4.8 TSpeedButton (eszköztár gomb, Additional paletta) 70
4.9 TCheckBox (jelölőnégyzet) 71
4.10 TRadioGroup (választógomb-csoport) 71
4.11 TListBox (listadoboz) 72
4.12 TComboBox (kombinált lista) 73
4.13 Menük használata 74
4.13.1 TMainMenu (Főmenü) 75
4.13.2 TPopupMenu (Gyorsmenü) 76
4.13.3 TMenuItem 76
4.14 Feladatok 77
4.14.1 „Sender" vadászat 77
4.14.2 Listadobozok 79
4.14.3 Vezérlőelem létrehozása futás közben 84
4.14.4 Mászkáló gomb (*TTimer* használata) 85
4.14.5 Tili-toli játék 88
4.15 Drag&Drop (fogd és vidd, vonszolás) technika 91
4.16 Egyéni kurzorok. A TScreen osztály 94
4.16.1 TScreen osztály 94

5. Több info az alkalmazásban 99

5.1 Fülek az űrlapon 99
5.1.1 TTabControl (Win32 paletta) 100
5.1.2 TPageControl (Win32 paletta) 100
5.1.3 TTabbedNotebook (Win 3.1 paletta) 101
5.1.4 TNoteBook (Win 3.1 paletta) 101
5.1.5 TTabset (Win 3.1 paletta) 101
5.1.6 A TNotebook és TTabset együttes használata 102
5.1.7 Feladat: „Csupafül" űrlap 103
5.2 Több űrlapos alkalmazások 104
5.2.1 Alkalmazásunk űrlapjai 104
5.2.2 Az ablakok megjelenítési formái 105
5.2.3 Egyszerű üzenet és adatbeviteli ablakok 106
5.2.4 Rendszer-párbeszédablakok használata 107
5.2.5 Az alkalmazások típusai 108
5.2.6 A TForm komponens 109
5.2.7 SDI alkalmazások készítése 114
5.2.8 MDI alkalmazások készítése 116
5.2.9 Feladat: MDI szövegszerkesztő írása 117

6. Grafika, nyomtatás 127

6.1 Tervezési időben létrehozható grafikai elemek 127
6.1.1 TShape (Additional paletta) 127
6.1.2 TImage (Additional paletta) 128

6.2 Futási időben létrehozható grafikai elemek 129
6.2.1 TCanvas osztály 129
6.3 Feladatok 133
6.3.1 Rajzolóprogram 133
6.3.2 Grafika listadobozban és fülsorban 138
6.3.3 Listaelemek grafikai összekötése (a *TList* osztály használata) 142
6.4 Nyomtatás 148
6.4.1 Nyomtatás a Printer objektummal 148
6.4.2 Szövegek nyomtatása 149
6.4.3 Grafikus ↔ karakteres nyomtatás 149
6.4.4 Feladat: szövegek és ábrák, karakteres és grafikus nyomtatása 151

II. RÉSZ: ADATBÁZISOK

7. Adatbázis-kezelés Delphiben 157

7.1 Az adatbázis-kezelési architektúrák áttekintése 157
7.1.1 Fájl-szerver (*File server*) architektúra 159
7.1.2 Kliens/szerver (*Client/Server*) architektúra 160
7.1.3 A több rétegű (*Multi-tier*) architektúra 162
7.2 A Delphi adatabázis-kezelési lehetőségei 163
7.3 Az álnév (*Alias*) 165
7.4 A Delphi adatbázis-kezelést elősegítő segédprogramjai 167
7.5 Adatbázis-kezelési komponensek 168
7.6 A TDataModule osztály 171
7.7 Feladat: Egy táblán alapuló böngésző 172

8. Adatelérési komponensek 175

8.1 Az adatelérési komponensek áttekintése 175
8.2 A TSession komponens 178
8.3 A TDatabase komponens 179
8.4 Az adathalmazok kezelése: TDBDataSet osztály 182
8.4.1 Adathalmazok állapotai 182
8.4.2 Adathalmazok nyitása, zárása 183
8.4.3 Mozgás az adathalmazban 184
8.4.4 Rekordok szerkesztése, törlése, új rekordok felvitele 187
8.4.5 Keresés az adathalmazban 190
8.4.6 Egy adathalmaz szűrése 194
8.4.7 Adathalmazok eseményei 199
8.5 Az adathalmazok mezői. A TField osztály 201
8.5.1 A mezőszerkesztő használata 202
8.5.2 Származtatott mezők létrehozása 205
8.5.3 A mezőobjektumok jellemzői, eseményei 210
8.5.4 Hivatkozás egy adathalmaz mezőire 214
8.6 A TTable komponens 216

8.7 A TDataSource komponens 220
8.8 Fő-segéd űrlapok készítése 222

9. Adatmegjelenítési komponensek 225

9.1 Az adatmegjelenítési komponensek használata 225
9.2 TDBGrid, TDBCtrlGrid 226
9.3 TDBNavigator 229
9.4 TDBListBox, TDBComboBox 229
9.5 TDBLookupListBox, TDBLookupComboBox 230

10. Feladat: Könyvnyilvántartó 233

10.1 Feladatspecifikáció 233
10.2 Az adatmodell 234
10.3 Az adatbázis létrehozása 237
10.4 Az alkalmazás űrlapjainak megtervezése 239
10.5 Az alkalmazás kivitelezése 245
10.5.1 Az adatmodul felépítése 245
10.5.2 Az űrlapok kivitelezése 246
10.5.3 Hibakezelés 261

11. SQL utasítások a Delphiben 269

11.1 Az SQL és a BDE 269
11.2 A TQuery komponens 270
11.3 A TQuery komponens használata 271
11.4 Az SQL utasítás megadásának módozatai 272
11.4.1 SQL megadása tervezéskor begépeléssel 273
11.4.2 SQL megadása tervezéskor a Database Desktop segítségével 274
11.4.3 SQL megadása a vizuális szerkesztővel (*Visual Query Builder*) 276
11.4.4 Az SQL megadása futásidőben 278
11.5 Paraméteres lekérdezések 279
11.5.1 A paraméter (-ek) megadásának módozatai 279
11.5.2 Feladat: Névböngésző kezdőbetű alapján 282

12. Feladat: A könyvnyilvántartó folytatása 285

12.1 Könyvek keresése témakör szerint 285
12.2 Egy könyv szerzőinek megszámlálása 288

13. Jelentések 291

13.1 A jelentések felépítése 291
13.2 A QuickReport komponenscsalád 293
13.3 A jelentések készítésének lépései 293
13.4 Jelentések példákon keresztül 294
13.4.1 Egyszerű jelentés létrehozása: vevők listázása 295
13.4.2 Csoportváltásos lista készítése: vevők kezdőbetűk szerint 297
13.4.3 Kétszintű csoportváltásos lista: vevők, megrendeléseik és tételeik. 299

13.4.4 Diagramok 304

14. Kliens/szerver adatbázis-kezelés egy feladaton keresztül 307

14.1 Feladatspecifikáció 307
14.2 Az adatbázis megtervezése 308
14.2.1 A fizikai adatbázis létrehozása 309
14.2.2 A mezőtípusok (*Domains*) létrehozása 310
14.2.3 A táblák létrehozása 311
14.2.4 A generátorok létrehozása 312
14.2.5 Pár szó a triggerekről és tárolt eljárásokról 313
14.2.6 A triggerek létrehozása 313
14.2.7 A tárolt eljárások létrehozása 317
14.2.8 A nézetek létrehozása 319
14.2.9 A jogosultságok beállítása 319
14.3 Az alkalmazás elkészítése 320
14.3.1 Az álnév létrehozása 320
14.3.2 Pár szó az alkalmazás-logikáról (*Business Logic*) 320
14.3.3 Az adatszótár (*Data Dictionary*) létrehozása 321
14.3.4 Az adatmodul felépítése 325
14.3.5 Az alkalmazás űrlapjainak megtervezése 326

III. RÉSZ: ÍNYENCSÉGEK

15. A komponensek fejlesztése 331

15.1 A komponensfejlesztés lehetőségei 332
15.2 TAlignButton 333
15.3 A komponenscsomagok fogalma 336
15.4 Komponens ikonjának beállítása 339
15.5 TIncCombo 341
15.6 TEnabEdit 343
15.7 TScrollList 345
15.8 TAboutBox 348
15.9 Súgó készítése egy saját komponenshez 351
15.10 Végszó 351

16. A súgó készítése 353

16.1 A súgó szerkezete és használata 353
16.2 A súgó készítésének lépései 355
16.3 Feladat: a könyvnyilvántartó súgójának elkészítése 356
16.3.1 A súgó szövegállományának (*.RTF) elkészítése 356
16.3.2 A súgó tartalomjegyzékének (*.CNT) elkészítése 359
16.3.3 A súgó projektállományának (*.HPJ) elkészítése 360
16.4 A súgó használata Delphi alkalmazásainkban 363
16.5 Tippek, tanácsok 364

17. A Delphi alkalmazások telepítése 367

17.1 Általános tudnivalók 367
17.2 Az InstallShield Express indítása 368
17.3 A telepítő külalaki adatai 369
17.4 A BDE állományainak kiválogatása 371
17.5 Az alkalmazás csoportjainak és állományainak megadása 373
17.5.1 Az alkalmazás állományainak megadása 374
17.5.2 A komponensek konfigurálása 375
17.5.3 Az általános, egyéni és minimális telepítés konfigurálása 376
17.6 A párbeszédablakok beállítása 376
17.7 A regisztrációs adatbázis bejegyzései 377
17.8 A program csoportjának és ikonjának beállítása 377
17.9 A telepítőkészlet létrehozása 377
17.10 Próbatelepítés 378
17.11 Mi változik az adatbázis-szerverek esetén? 378

18. Az alkalmazások közötti kommunikáció 379

18.1 A vágólap (*Clipboard*) használata Delphiben 379
18.2 A DDE (*Dynamic Data Exchange*) technika 380
18.2.1 DDE kliens alkalmazás készítése Delphiben 381
18.2.2 Hálózatos DDE kapcsolat (*NetDDE*) 385
18.3 Az OLE (*Object Linking and Embedding*) technika 387
18.3.1 OLE 1.0, OLE 2.0, OLE automatizmus 388
18.3.2 OLE automatizmus Delphiben 389

19. Több rétegű (*multi-tier*) adatbázis-kezelés 391

19.1 Feladatspecifikáció 391
19.2 A középső réteg elkészítése 393
19.3 A kliens alkalmazás elkészítése 396
19.4 Végszó 399

20. Több szálon futó alkalmazások 401

20.1 A szál (*thread*) fogalma 401
20.2 Több szálú alkalmazások a Delphiben 402
20.3 Több szálú adatbázisos feladat 403
20.3.1 Az adatmodul megtervezése 403
20.3.2 A szálak megtervezése 404
20.3.3 Az űrlap megtervezése 406
20.3.4 A feladat Interbase-es megvalósítása 409

Irodalomjegyzék 412

Tárgymutató 413

I. rész
Alapfogalmak

E rész tanulmányozásához szükséges előismeretek

Egy Delphi program megértése, céltudatos megírása rengeteg előismeretet feltételez: ismernünk kell a konkrét nyelv (Object Pascal) elemeit és sajátosságait, az alkalmazások szerkezetét, a Windows eseményvezérelt technikáját és nem utolsósorban a konkrét fejlesztőkörnyezet lehetőségeit.
Ebben a részben a **Windows és Delphi** programozásának alapfogalmairól lesz szó, így hát erősen támaszkodunk a következő ismeretekre:

- A Turbo Pascal nyelv szintaxisa
- Az objektumorientált programozás és tervezés alapjai — UML.

Mit nyújt ez a rész Önnek?

- **1. fejezet: Windows bevezetés**

  Áttekintjük a Windows rendszer általános jellemzőit, majd programozói szemszögből beszélünk az eseményvezérelt technológiáról.

- **2. fejezet: Delphi bevezetés**

  Megismerkedünk a Delphi alkalmazások szerkezetével, és elkészítjük első alkalmazásunkat.

- **3. fejezet: A Turbo Pascaltól az Object Pascalig**

  Áttekintjük az Object Pascal nyelvben bevezetett újdonságokat, mint például az új osztálymodellt, a kivételek kezelését, a karakterlánc-típusokat stb.

- **4. fejezet: Standard komponensek**

  Megismerkedünk a leggyakrabban használt vezérlőelemekkel. A fejezet második felében számos feladaton keresztül be is gyakorolhatjuk eddigi ismereteinket. Ugyanitt esik szó a *drag&drop* technikáról, valamint az egyéni kurzorformák alkalmazásáról.

- **5. fejezet: Több info az alkalmazásban**

  Megtanulhatjuk adatainkat egy űrlapon több oldalra elhelyezni, majd megismerkedhetünk a több ablakos alkalmazások készítésének módjával. Bemutatjuk az SDI és MDI alkalmazások szerkezetét elméletben és a gyakorlatban is.

- **6. fejezetben: Grafika, nyomtatás**

  Megismerkedünk a Delphi grafikai és nyomtatási lehetőségeivel elméletben, majd feladatokon keresztül is.

1. Windows bevezetés

Az általános felhasználói réteget tekintve állíthatjuk, hogy manapság a PC-k világát éljük, a PC-ken pedig legtöbbször Windows operációs rendszer fut. A windowsos világ előnyeit mindnyájan ismerjük: lehetővé teszi az egymással kommunikáló programok párhuzamos futtatását, valamint az alkalmazások szabványos – ablakokra alapozott – grafikai megjelenítését. Ebben a fejezetben az 1.1. pontban a Windows rendszerek főbb jellemzőit fogjuk áttekinteni; e pont elolvasását feltétlenül javaslom minden kedves Olvasómnak. Az 1.2. pontban az eseményvezérelt programozás megvalósítási módjával ismerkedhetünk meg. Ez a Windows programozásával most ismerkedők számára kicsit „mélyebb vizet" jelenthet. Ha így lenne, akkor se csüggedjen el, folytassa a 2. fejezettel, majd később olvassa újra, ha már nagyobb tapasztalattal rendelkezik. Olyan – talán kevésbé ismert – háttérjelenségekkel, mechanizmusokkal ismerkedhetünk itt meg, melyek hiányában egy programozó még egy 4GL nyelvben sem boldogul, nem képes megérteni a pillanatokon belül elkészített alkalmazás működését, és ebből kifolyólag javításokat sem képes rajta eszközölni.

1.1 Általános tudnivalók

A Windowsnak ma már több változatát ismerjük. Az első általánosan elterjedt változat, a 16 bites Windows 3.1, még nem teljesen különálló operációs rendszer, hiszen a DOS-ra épül. A későbbi változatok már 32 bitesek, ezek a Windows 95 (Win95) és a Windows NT (WinNT). A WinNT már különálló operációs rendszer, a Win95 pedig szerkezetileg a Win 3.x és a WinNT között található.

A Windows rendszer jellemzői:

- **Multitasking**

  Windowsban párhuzamosan több alkalmazást is futtathatunk. Ezek lehetnek különbözőek (pl. Word és Excel), de lehetnek egyazon alkalmazás több példányai is (pl. Word két példánya, mindegyik a saját ablakában). A Windows 16 és 32 bites változataiban a *multitaskingnak* két különböző típusával találkozunk: a *non-preemptive* és a *preemptive multitaskinggal*. Bővebben lásd az 1.2.4. pontban.

- **Dinamikus adatcsere lehetőségek**

  Windowsban a párhuzamosan futó alkalmazások képesek egymással kommunikálni. A kommunikációnak több módja van:

  - **Vágólap (*Clipboard*)**

    A vágólap egy Windows szinten közös memóriaterület, melyet minden futó alkalmazás használhat. Egy alkalmazásból szövegrészeket, táblázatokat, képeket stb. másolhatunk vagy vághatunk ki erre a területre, majd ezeket később – akár több példányban is – beilleszthetjük más alkalmazásokba. Például egy Excel táblázatrészt vágólapon keresztül áthelyezhetünk a Word dokumentumunkba.

  - **DDE (*Dynamic Data Exchange* = Dinamikus adatcsere)**

    A DDE leginkább szöveges információk (szövegek vagy makróparancsok) cseréjét teszi lehetővé a párhuzamosan futó alkalmazások között egy ún. kommunikációs csatornán keresztül. Például egy Delphi alkalmazásból küldhetünk egy Excel munkalap cellájába egy szöveget, és ennek tartalmát onnan vissza is olvashatjuk. Ugyancsak DDE kommunikációs csatornán keresztül egy szöveges makróparanccsal utasíthatjuk a Windows programkezelőjét (*Program manager*) például arra, hogy egy programcsoportot hozzon létre. (Ezt általában a telepítőprogramok végén szoktuk kérni.)

  - **OLE (*Object Linking and Embedding* = Objektumok csatolása és beágyazása)**

    Az OLE technikának köszönhetően alkalmazásainkat olyan szolgáltatásokkal is feldúsíthatjuk, melyeket más alkalmazások bevonásával fognak kielégíteni: alkalmazásunk beágyazva vagy csatolva tartalmazhat egy másik program által létrehozott objektumot. Például egy Winword dokumentumba elhelyezhetünk egy PaintBrush képet; ha szerkeszteni szeretnénk a képet, elég rá duplán kattintanunk, és máris betöltődik a PaintBrush, elkezdődhet a szerkesztés. Az OLE technikának több változata van (OLE 1.0, OLE 2.0, OLE automatizmus), ezekről, valamint a többi adatcsere lehetőségről is részletesebben a 18. fejezetben lesz szó.

- **Memóriakezelés (*Windows Memory Management*)**

  A memória nagyon fontos erőforrás[1] a Windows környezetben, hiszen kezelésétől függ a rendszer hatékonysága. Tekintsük át röviden a Windows memóriakezelésének fő alapelveit:

---

[1] Windowsban erőforrásnak nevezünk minden – a program működéséhez szükséges – szoftver, illetve hardver eszközt, például memória, processzor, háttértároló, fontok, egérkurzor (formája *.CUR állományban tárolódik), ikon (*.ICO), súgóállomány (*.HLP) stb.

- A *multitasking* csak megfelelő memóriakezeléssel valósítható meg. Ugyanazon alkalmazás párhuzamosan futó példányai közös kódblokkokat használnak, de ugyanakkor külön adatblokkokkal rendelkeznek.
- A windowsos alkalmazásaink által lefoglalt memóriaterületek (programjaink kód- és adatblokkjai) általában a memória területén belül mozgathatók (*moveable*), valamint eldobhatók (*discardable*), azaz szükség esetén eltávolíthatók a memóriából. A kódblokkokat a rendszer csak szükség esetén tölti be a memóriába, akkor, amikor valamely futó programnak szüksége van rájuk. Minden új blokk betöltésének előfeltétele az, hogy rendelkezésre álljon megfelelő méretű szabad hely a tárban, azaz a betöltést egy helyfelszabadításnak kell megelőznie. A ***„Least Recently Used"*** (***LRU*** = „legkevésbé frissen", azaz legrégebben használt) elv alapján a Windows a legrégebben – azaz a legkevésbé – használt blokkot fogja eltávolítani a memóriából, és ennek helyére fogja az új blokkot beolvasni. Ezt nevezzük virtuális memóriakezelésnek vagy *swap* technikának.

Ezen elvek lehetővé tették, hogy Windowsban a fizikai memória kapacitásánál nagyobb programokat is futtathassunk, sőt azt is, hogy több nagyméretű alkalmazást párhuzamosan elindíthassunk.

- **Dinamikus könyvtárkezelés**

A Windows rendszer talán legjelentősebb strukturális elemei a dinamikusan szerkeszthető rutinkönyvtárak (***Dynamic Link Library***-k). A DLL-ek nem végrehajtható állományok, csupán rutinkönyvtárak, melyeknek elemeit (függvényeit és eljárásait) bármely Windows alkalmazás használhatja. Ezek a rutinok DLL, EXE[1] és egyéb (például OCX) kiterjesztésű állományokban találhatók. Számos DLL már a Windows rendszernek része (pl. KERNEL.EXE, USER.EXE, GDI.EXE...). Ugyanakkor készíthetünk saját DLL-eket is, ezzel kiegészítve a Windows rendszert újabb funkciókkal. A legfontosabb Windows DLL-ek szerepét az 1.1. ábra mutatja.

Könyvtár	Szerepe
KERNEL32	– memóriakezelés – programok betöltése, kezelése
USER32	– ablaktechnika, felhasználói felület
GDI32[2]	– grafikus megjelenítés

1.1. ábra. A 32 bites Windows főbb rutinkönyvtárai és azok szerepe
(16 bites Windowsban: KERNEL, USER, GDI)

---

[1] Ez egy speciális EXE formátum, melyben egy összesítő táblázatban megtalálható minden könyvtárbeli rutin címe.

[2] GDI = **G**raphics **D**evice **I**nterface

A Windows programokból hívható rutinok együttes neve az **API** (*Application Programming Interface*), ezek a különböző DLL-ekben található függvények és eljárások.

- **Grafikus felhasználói felület** (*Graphical User Interface*)

  A grafikus ablakalapú felhasználói felületet először 1970-ben a PARC (Palo Alto Research Center) fejlesztette ki, majd 1984-ben az Apple is átvette, alkalmazta. Főbb előnyei:

  - **Felhasználóbarátság**

    Erre vonatkoznak a *„WYSIWYG"* (*What You See Is What You Get*) és a *„Look and Feel"* elvek is, azaz az ablakban megtalálható minden elérhető funkció. Elég kattintanunk, és máris érezhető, látható az eredmény. A felhasználóbarátsághoz az is hozzátartozik, hogy minden ablakban egységes erőforrásokat találunk: menüsorok, gombok, ikonok, párbeszédablakok stb. Ily módon a Windows rendszer felhasználójának igazán könnyű dolga van.

  - **Windows betűtípusok**

    A betűk kijelzése Windowsban nem karaktergenerátor segítségével történik (mint DOS-ban), hanem ***raster***, ***vektor*** és ***TrueType*** technikával. Egy adott betűtípus esetén a betűk külalakja állományokban van letárolva, akár bittérkép formájában (*raster*), akár egy algoritmus formájában, melynek segítségével megrajzolható az illető betű (*vektor* és *TrueType*). Ennek köszönhető az a tény, hogy szövegeink szerkesztésekor oly sokféle betűtípust használhatunk.

  - **GUI ↔ hardver függetlenség**

    Windowsban óriási előnynek számít az, hogy a felhasználói felület teljesen független a konkrét hardver konfigurációtól. A programozó egységesen kezelheti a különböző I/O eszközöket: billentyűzetet, egeret, képernyőt, nyomtatót stb. A különböző eszközmeghajtókkal (*driver*) nem nekünk kell törődnünk, ez a Windows rendszer feladata. Mindezek a szabványos grafikus könyvtár rutinjainak segítségével valósíthatók meg (**GDI** = ***Graphics Device Interface***). Ez a könyvtár a Windows rendszer része (lásd 1.1. ábra).

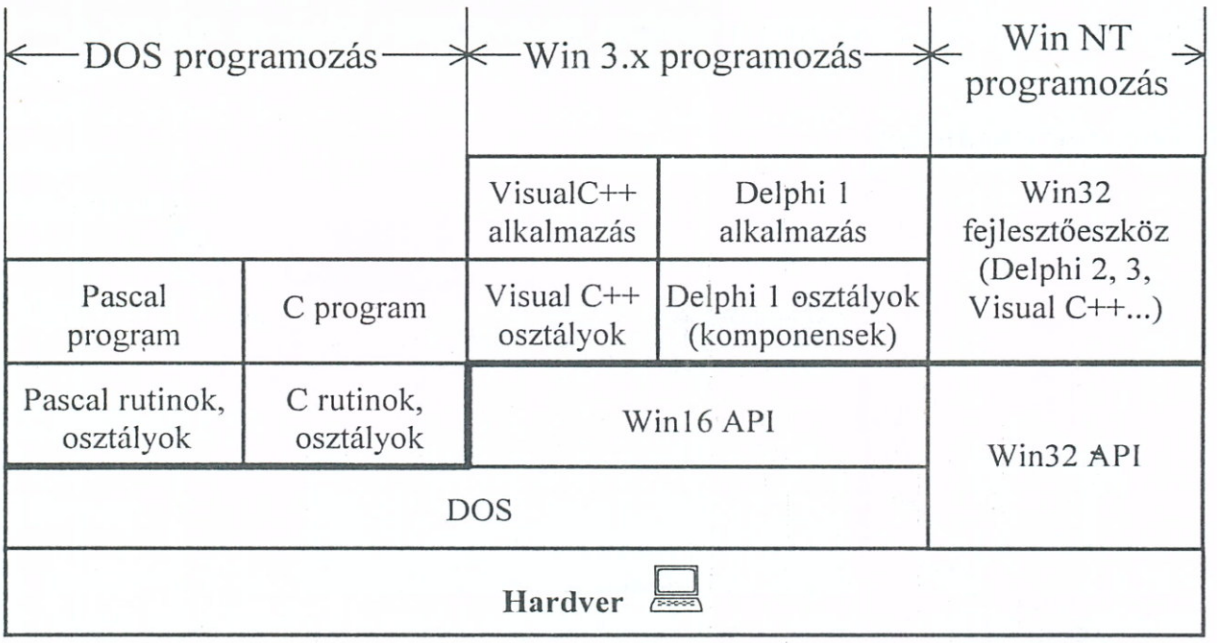
- **DOS ↔ Windows kapcsolat**

  A Windows 3.x változatai még nem működnek külön operációs rendszerként, sok mindenben támaszkodnak a „jó öreg" DOS-ra. Ezekben a rendszerekben néhány feladat továbbra is a DOS-ra hárul (pl. az állományok kezelése), míg másokat átvállalt a Windows (képernyő, billentyűzet, egér, nyomtató, portok, memória kezelése, programok betöltése és kezelése). A Win95-ben még többé-kevésbé szükség van a DOS-ra, a WinNT-ben azonban már minden feladatot a Windows lát el (lásd 1.2. ábra).

  Ha WinNT előtti Windows verziót használunk, akkor megfordulhat fejünkben a következő kérdés: vajon nem írhatunk-e egy hagyományos DOS-os programot úgy, hogy csak a megjelenítést és egérkezelést vegyük át a Windowstól? Természetesen

NEM. Egyrészt ez a perifériák kezelése miatt lehetetlen, másrészt a windowsos és DOS-os programok között elvi különbségek is vannak. Míg DOS-ban a program írója mondja meg, hogy ez mikor, milyen adatot kér be és ír ki, addig a Windowsban a felhasználó az, aki „össze-vissza" kattintgat, és kénye-kedve szerint szabályozza az alkalmazás futását. Valójában azt lehet mondani, hogy egy windowsos program nem tesz mást, mint megjeleníti az ablakait, és ezek után nagy figyelemmel várja a felhasználó akcióit. Ezek után a felhasználó szabályozza alkalmazásunk futását a különböző gombok, menüpontok stb. hívásával.

Egy windowsos program minden részének windowsos elveken kell működnie.



1.2. ábra. Programozás DOS-ban és Windowsban
(A Win95 szerkezetileg a Win 3.x és a WinNT között található.)

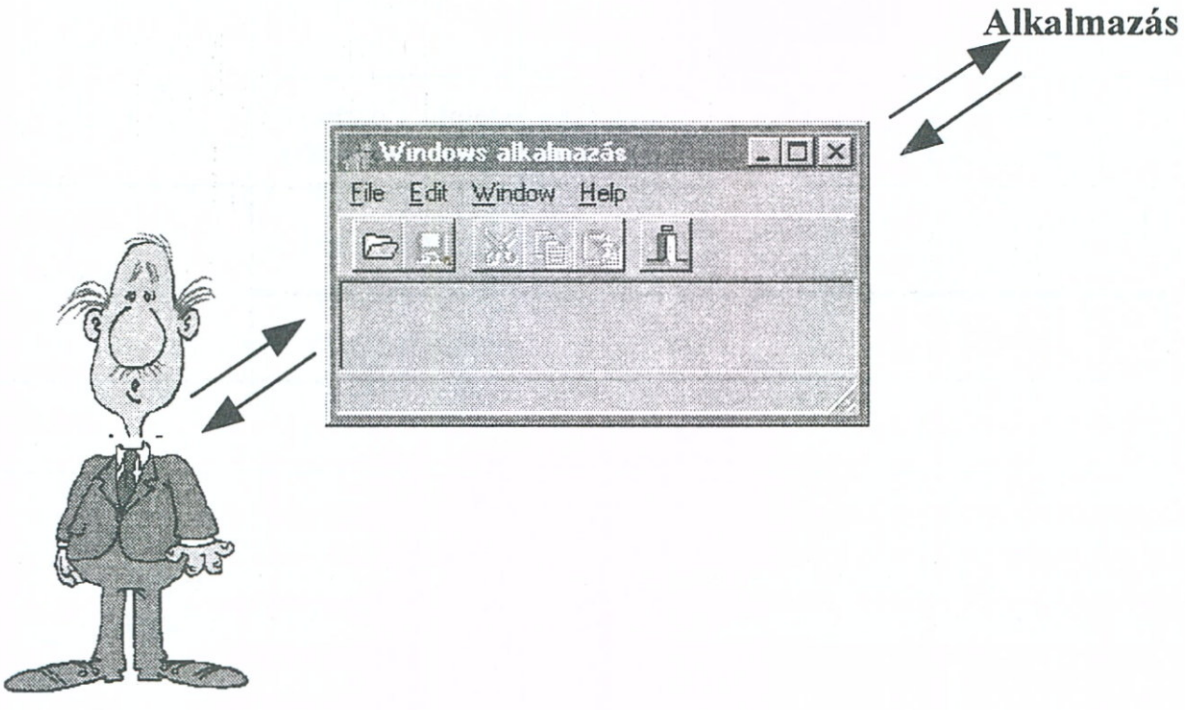
DOS-os környezetben programozhatunk Pascalban, C-ben, vagy egyéb magas szintű nyelvekben. A kiválasztott nyelv ráépül az operációs rendszerre, mintegy kibővítve azt egy rutincsomaggal. Az így fejlesztett programokban hívhatjuk a nyelv rutinjait (esetleg, ha az objektumorientált, akkor objektumainak metódusait is), hívhatunk megszakításokat, és, ha nagyon akarjuk, kezelhetjük közvetlenül is a hardvert (pl. képernyőmemóriába való közvetlen írás).

Windowsban az absztrakciós szint megemelkedett. Maga a Windows rendszer eleve kibővíti a DOS-t egy API-nak nevezett rutincsomaggal. Ezek a rutinok minden windowsos fejlesztő környezetből hívhatók. Erre ráépülnek az egyes környezetek rutincsomagjai vagy osztálykönyvtárai, attól függően, hogy az adott környezet mennyire

objektumorientált. Egy Delphi programban hívhatjuk a Delphi objektumok metódusait, és – ha ez nem elég, akkor – rendelkezésünkre áll az API több ezer rutinja.

A DOS-szal ellentétben a Windowsban már „nem illik" megszakításokat hívni, vagy közvetlen hardverkezelést lebonyolítani. Ezt a Windows rendszerek egyre kevésbé támogatják. Sőt! Az alkalmazásunk akkor lesz csak igazán hordozható, ha abban még API hívások sincsenek, mivel ezek formája is változhat a különböző Windows verziókban. Ha például 32 bites Windows környezetben elindítunk egy 16 bites alkalmazást (mely természetesen 16 bites API rutinhívásokat tartalmaz), akkor ez beránt maga mellé egy „fordítót", mely a 16 bites API hívásokat 32 bites API-vá alakítja. Ezért tapasztaljuk azt, hogy a 32 bites környezetekben a 16 bites alkalmazások lassúbbak, mint ha ugyanazt az alkalmazást 32 bitesre lefordítanánk. Az API hívásokat mellőző Delphi program forrásszinten válik hordozhatóbbá, azaz egy 16 bites Delphiben írt alkalmazást lefordíthatunk 32 bites környezetben, és így egy gyorsabb (32 bites) futtatható állományt kapunk, mint a 16 bites környezetben generált változata.

- **Objektumorientáltság**



1.3. ábra. A felhasználó egy ablakon keresztül kommunikál az alkalmazással

A Windows rendszer alapvetően objektumorientált. Minden alkalmazása ablakokban jelenik meg. Minden ablak objektumként fogható fel, hiszen rendelkezik adatokkal és viselkedéssel.

- Adatai: az ablak címe, menüje, színe, gombjai, kurzor formája, ikonja stb.
- Viselkedése: reakciója a különböző külső és belső eseményekre

Amint ezt az 1.3. ábra is mutatja, a felhasználó az ablak-objektumon keresztül kommunikál az alkalmazással. Az ablakon keresztül kérhet szolgáltatásokat, és az ablakban fogja látni ezek eredményeit.

Az ablak és a felhasználó közti információcsere, a teljes rendszer működése az üzenetek közvetítésén alapul.

Más objektum-jelöltek is vannak a Windows rendszerben:

- Ha egy windowsos alkalmazásban nyomtatni szeretnénk, akkor ezt a nyomtató-objektum segítségével tehetjük meg. Ez egy tipikusan szerver objektum, ő szolgálja ki a rendszerből érkező és nyomtatással összefüggő kéréseket.
- A legtöbb windowsos alkalmazásban az állomány megnyitási, mentési, nyomtatási, betűtípus beállítási feladatokat ugyanazokkal a párbeszédablakokkal valósítják meg. Például Wordben és Excelben ugyanúgy néznek ki a *File/Open*, *Save*, *Save as...*, *Print* menüpontok hatására megjelenő ablakok. A párbeszédablakok is objektumok, melyek a Windows rendszer részei, innen „veszi kölcsön" a Word, Excel, Delphi... Ezek az objektumok rendelkeznek megjelenési tulajdonságokkal, valamint tipikus viselkedéssel is (pl. mi történik, ha az *OK* vagy a *Cancel* gombokra kattintunk).

1.2 Windows eseménykezelés, üzenetvezérlés

1.2.1 Az esemény és az üzenet fogalma

Esemény (Event)

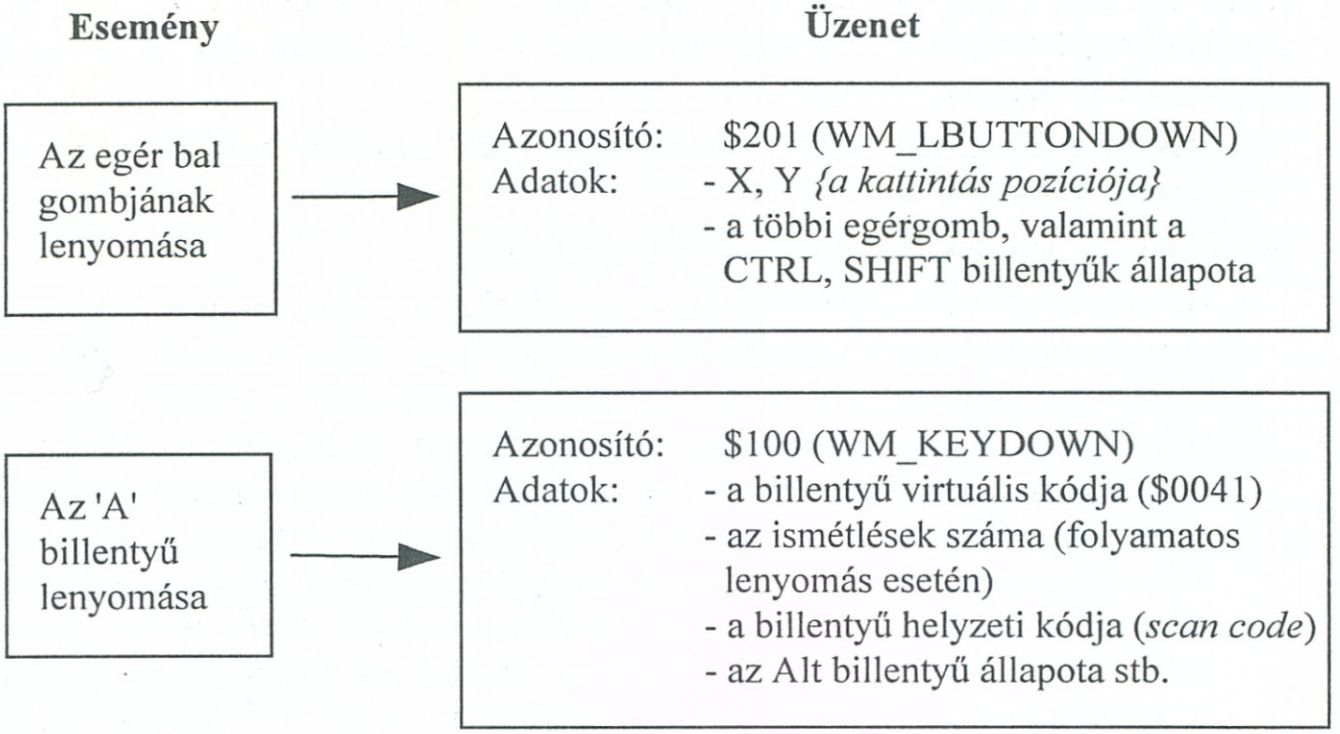
Amikor egy program felhasználója kérni szeretne valamit a programtól, akkor a billentyűzet, az egér, vagy más beviteli eszköz segítségével hat a rendszerre. Bekövetkezik tehát egy esemény. Ezt a „megfoghatatlan" külső történést a Windows rendszer fogadja, és üzenetté alakítja.

Üzenet (Message)

Az üzenet egy rekord, melyet a Windows rendszer állít elő, és adatai az őt kiváltó esemény típusára, paramétereire vonatkoznak. Típusa a *TMessage*:

```
Type TMessage = Record
       Msg: Word;          {az üzenet azonosítója}
       wParam: Word;       {az üzenet paraméterei egy Word...}
       lParam: LongInt;    {és egy Longint típusú mezőben}
       Result: LongInt;    {az üzenet feldolgozásának eredménye}
     End;
```

Nézzük meg például, milyen üzenet keletkezik akkor, amikor kattintunk az egér bal gombjával vagy amikor lenyomjuk az 'A' billentyűt (1.4. ábra):



1.4. ábra. Esemény átalakítása üzenetté

A billentyűzetről érkező eseményeknél a Windows az üzenetben a billentyű úgynevezett virtuális kódját (*virtual-key code*) tárolja, mely egyértelműen azonosítja a lenyomott billentyűt. Ez a kód egy Windows rendszerbeli konstans érték[1]: az 'A' esetén *VK_A* = $41[2]; 'B' esetén *VK_B* = $42; 'Delete' estén *VK_DELETE* = $2E stb.

Az üzenetek azonosítói konstansként is szerepelnek a rendszerben, így a konkrét számértékekkel nem kell törődnünk.
Például:

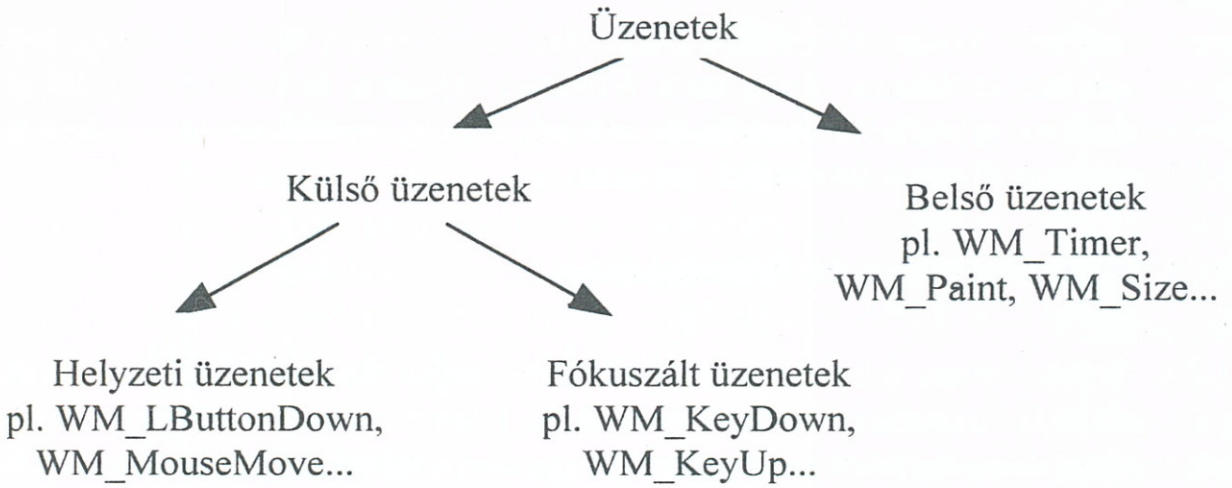
WM_LBUTTONDOWN = $201 (***Windows Message Left Button Down***)
WM_KEYDOWN = $100
WM_KEYUP = $101

---

1 Lásd API súgó (WIN32.HLP) *Virtual-Key Codes* témakör.

2 A $ jel – Pascal jelölés szerint – azt jelenti, hogy a szám a 16-os számrendszerben értendő.

1.2.2 Az üzenetek típusai



1.5. ábra. Üzenetek típusai

Minden eseményből legalább egy üzenet képződik a Windows rendszer „bejáratánál". Ezeket **külső üzenetek**nek nevezzük, mivel valamelyik perifériáról érkeznek a rendszerbe. A külső üzenetek kétfélék lehetnek: **helyzeti**ek vagy **fókuszált**ak. Helyzetinek nevezünk egy üzenetet akkor, ha fontos jellemzője bekövetkezésének képernyőpozíciója (pl. egérgomb kattintásakor). A fókuszált üzenetre nem jellemző a képernyőpozíció; ha lenyomjuk az 'A' billentyűt, akkor minket a billentyű (billentyűzeten való) helyzeti kódja[1] (*scan code*) fog érdekelni, nem pedig az egérkurzor pillanatnyi pozíciója.
Üzenetek a rendszeren belül is születhetnek. Ha egy ablakot elmozdítunk vagy átméretezünk, akkor a frissen felderített részt újra kell rajzolnunk; ez egy **belső üzenet** – a *WM_PAINT* – hatására automatikusan megtörténik. De belső üzenet keletkezik akkor is, ha egy belső hiba áll elő, ha érkezik egy órajel (*WM_TIMER*) stb.
Az üzeneteket (származásuktól függetlenül) a Windows rendszer megfelelő szabályok alapján szétosztja a futó alkalmazások között.
A fókuszált üzenetet a fókuszban levő ablak fogja megkapni és feldolgozni úgy, ahogy azt az ablak tervezői elgondolták.
A helyzeti üzenetet általában az az alkalmazás, illetve az alkalmazásnak az az ablaka kapja, amelynek a felületén történt. Előfordulhat persze az is, hogy átkattintunk egy másik alkalmazás ablakába. Ilyenkor az eddigi aktív alkalmazás háttérbe szorul (mert kap egy ezt előidéző belső üzenetet), az új alkalmazás pedig előbukkan a háttérből, takart részei kirajzolódnak. Ettől a pillanattól kezdve ez lesz az aktív alkalmazás. Ebben az esetben tehát nem csak az az alkalmazás kapott üzenetet, amelyikre kattintottunk, hanem egy másik is. Lám-lám, mennyi minden történt, holott mi csak egyet kattintottunk. Ez egy külső üzenet-

---

[1] A billentyűzet minden billentyűjéhez egyedi helyzeti kód tartozik. A rendszer ebből állítja elő a billentyűhöz tartozó betű kódját az aktuális billentyűzetkiosztásnak (angol, magyar) megfelelően.

ben nyilvánult meg, a többit a Windows rendszer hozta létre belső üzenetként. Ezt nevezzük **üzenetlánc**nak.
Természetesen most nem kívánjuk beleásni magunkat az üzenetek létrehozásának rejtelmeibe. Ebbe túl sok beleszólásunk nincs, ez a Windows feladata, és feltehetőleg jól el is látja. Fontos azonban tudni ezek létezéséről, és szükség esetén ezeket nyomon is kell tudnunk követni. Erre használható a WinSight[1] program.

1.2.3 Üzenetek szétszórása és feldolgozása

Általános fogalmak:

- A 16 bites Windowsban minden futó alkalmazás **saját üzenetsor**ral rendelkezik, melyben a neki szánt üzenetek kerülnek bele. A 32 bites windowsos alkalmazásokban már több szálon (*thread*) futhat a program[2], itt minden szálnak külön üzenetsora van. Ezekbe a sorokba a rendszer helyezi el az üzeneteket, ez válogatja és osztja szét a párhuzamosan futó alkalmazások között a bekövetkezett események üzeneteit. A 16 bites Windowsban a külső üzenetek nem kerülnek közvetlenül az alkalmazások soraiba, hanem ezeket a rendszer ideiglenesen egy rendszerszintű üzenetsorba gyűjti ösz-sze. A 32 bites Windowsban már nincs rendszerszintű üzenetsor (ez nyilván a *preemptive multitasking*-gal magyarázható, lásd később az 1.2.4. pontban), így az üzenetek egyből az alkalmazások (szálak) soraiba kerülnek.

De hogyan kerülnek az üzenetek a megfelelő sorokba?
A Windows bejáratánál keletkezett üzeneteket a rendszer sorban megvizsgálja. Minden üzenetről eldönti, hogy mely alkalmazásoknak, azon belül mely szálaknak, és milyen formában kell továbbítania. Az *Alt* + *Tab* billentyűkombináció valószínűleg több alkalmazás működését is befolyásolni fogja, míg egy egyszerű 'A' billentyű leütése üzenetének csak az aktív ablakhoz kell eljutnia. Ha viszont egy üzenet bekerült egy alkalmazás valamelyik szálának sorába, akkor ettől a pillanattól kezdve a Windows már nem tehet semmit, most már az alkalmazáson a sor, neki kell a beérkezett üzenetet valamilyen módon feldolgoznia.

- A windowsos alkalmazások minden szálában megtalálható egy **„üzenetkezelő ciklus"** (***message loop***). Feladata az, hogy kiolvassa a sorban álló üzenetek, majd továbbítsa ezeket a megfelelő ablakhoz. Körülbelül így:

```
Ciklus amíg nincs vége az alkalmazásnak (szálnak)
  Üzenet kiolvasása a sorból {GetMessage}
```

---

[1] Ez a program a legtöbb Windows-os fejlesztőeszközzel együtt érkezik, megtalálható a Delphi csoportjában is. Segítségével megfigyelhetjük, hogy milyen üzenetek képződnek a rendszerben, mely ablak fogadja, és hogyan dolgozza fel ezeket.

[2] Egy alkalmazásban a több, párhuzamos futó szálat leginkább úgy lehet elképzelni, mint több kicsi alkalmazást egy nagyobbon belül. A szálakról bővebben a 20. fejezetben lesz szó.

```
  Üzenet esetleges átalakítása {TranslateMessage}
  Üzenet továbbítása a megfelelő ablakfüggvénynek
  {a Windowson keresztül: DispatchMessage}
Ciklus vége
```

Egy billentyű leütésekor az üzenetbe annak virtuális kódja kerül. Később azonban a betű kirajzolásakor szükség van annak ASCII kódjára is. Ezt az átalakítást (virtuális kód → ASCII kód) az üzenetkezelő ciklusban kell elvégezni (egy speciális API függvénnyel: *TranslateMessage*). Az átalakításnak természetesen csak akkor van értelme és hatása, ha a billentyűnek tényleg megfelel egy ASCII kód (például egymagában a *Shift* billentyűnek nincs ASCII kódja).

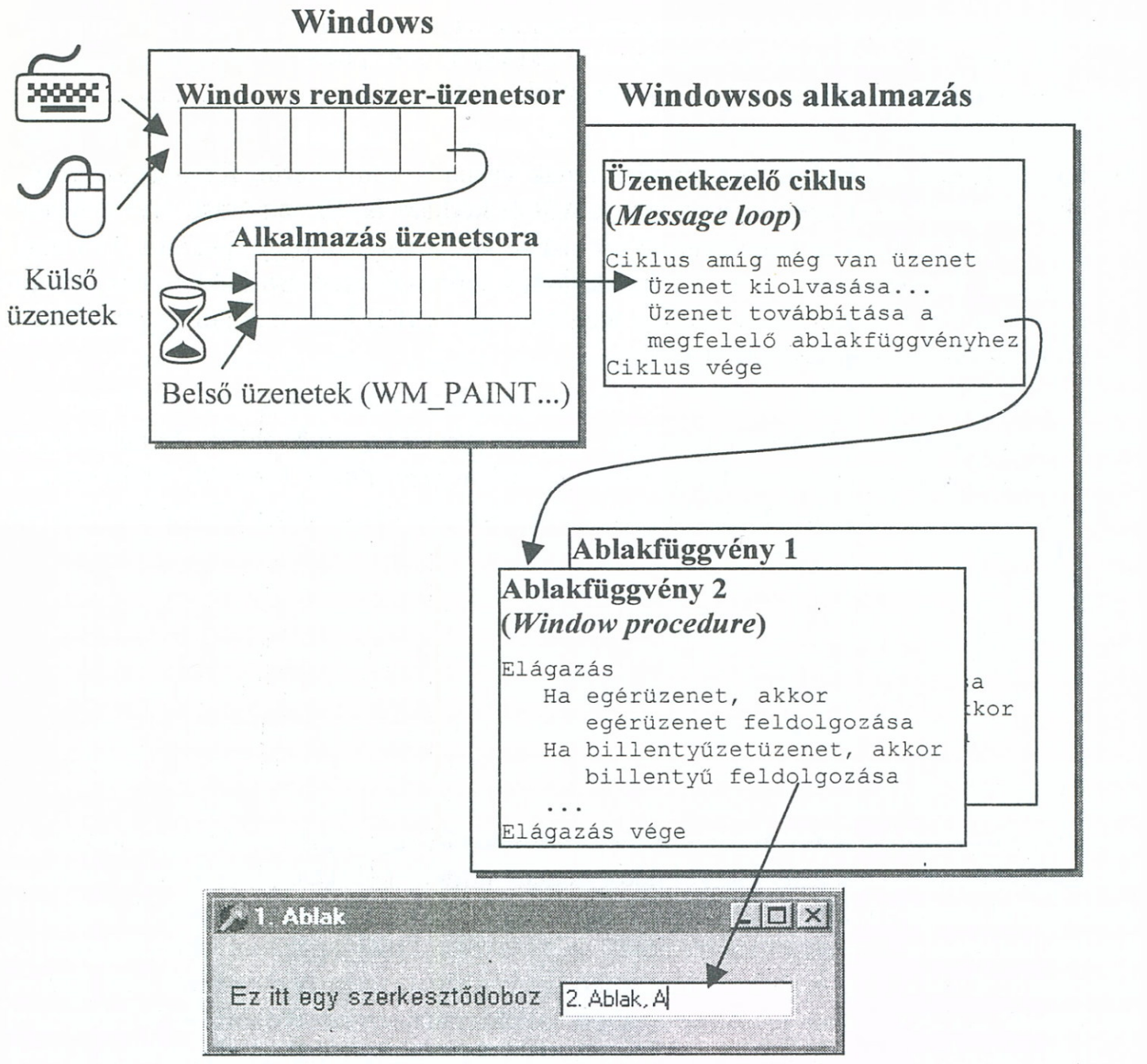
- Minden alkalmazásban van legalább egy, de általában több ablak. Minden egyes ablaknak van egy saját **ablakfüggvénye** (***window procedure***), melyben feldolgozza a neki szánt üzeneteket. Ez egy hatalmas elágazás, mely logikailag a következőképpen néz ki:

```
Elágazás
  Üzenet = Kattintás esetén
     Kattintás feldolgozása
  Üzenet = Billentyű lenyomás esetén
     Billentyű lenyomásának feldolgozása
  ...
Elágazás vége
```

Programunkban tehát minden ablak rendelkezik egy saját ablakfüggvénnyel. Az ablakokat az alkalmazás elindításakor regisztrálnunk kell (Delphiben ezt a rendszer végzi el, de például Borland C++ 3.1-ben még nekünk kellett kódolnunk). A regisztrálás hatására a rendszer „megismeri" az ablakokat, megjegyzi az ablakfüggvények címeit. Ezek a függvények közvetett módon hívódnak meg. A megfelelő szál üzenetkezelő ciklusa utolsó tevékenységeként az üzenetet „visszadobja" a Windowsnak (*DispatchMessage*), aki – az erre vonatkozó ismeretek birtokában – meghívja a megfelelő ablakfüggvényt.

Egy windowsos alkalmazás lényegében az ablakfüggvények elágazásaiban implementált rutinokból áll, ezek tartalmazzák a feladatspecifikus feldolgozásokat.

Ennyit általánosságban, és most nézzünk egy konkrét példát: kövessük végig az 'A' billentyű lenyomásának feldolgozását. Az 1.6. ábrán Windows rendszer-üzenetsort és alkalmazás üzenetsort láthatunk. Ez a 16 bites rendszerekben igaz. A 32 bites Windowsban annyi változik, hogy ott nincs rendszerszintű üzenetsor, és egy alkalmazáson belül minden szálnak saját sora van.



1.6. ábra. Az események feldolgozása a 16 bites Windowsban

A Windows rendszer fogadja „az 'A' billentyű leütése" eseményt, azonnal üzenetté alakítja, majd elhelyezi a rendszerszintű üzenetsorban. Itt megvizsgálja az üzenetet, tapasztalja, hogy ezt csak az aktív alkalmazásnak kell továbbítania (ez más alkalmazásokat nem érint), tehát meg is teszi. Most már az applikáción a sor. A főprogramjában levő üzenetkezelő ciklusban kiolvassa, átalakítja, majd a rendszer segítségével közvetett módon eljuttatja a megfelelő ablakfüggvényhez. Igen ám, de vajon melyik ablak fogja az üzenetet megkapni? Fókuszált üzenetről lévén szó, a fókuszban levő ablak a címzett. A mi esetünkben ez a szerkesztődoboz (2. Ablak), tehát az ő ablakfüggvényét kell meghívni (paraméterében átveszi az üzenetet). A szerkesztődoboz valószínűleg úgy fogja a billentyűzet üzenetét feldolgozni, hogy a karaktert megjeleníti a soron következő pozíciótól. Igen ám, de a

kiíráshoz is API (pontosabban GDI) függvényre van szükség. Így hát „a labda hosszas ide-oda dobálásával" végre megszületett az eredmény, az új betű megjelent a szerkesztődobozban, a kurzor pedig ott villog utána. Az alkalmazás készen áll a további üzenetek fogadására. Ebből azt a következtetést vonhatjuk le, hogy ameddig az egyik üzenet feldolgozása folyik, addig a többi – később bekövetkezett – üzenet kényszeredetten várakozik a sorban.

Vegyünk egy másik példát: tegyük fel, hogy írtunk egy telepítőprogramot. Ezt elindítjuk, beállítjuk a célkönyvtárat (a telepítendők helyét), majd a *Telepítés* gombra kattintunk. A gomb hatására (*WM_LBUTTONDOWN*) megkezdődik a telepítendő állományok átmásolása. Ez a folyamat általában több időt vesz igénybe. Ha a felhasználó közben meggondolja magát, és ki szeretne lépni a telepítésből, akkor feltehetőleg a *Mégsem* gombra kattint. Vajon mi történik ilyenkor? Összesen két külső üzenet érkezik alkalmazásunkhoz: az első a *Telepítés* gombra, a második a *Mégsem* gombra való kattintás üzenete. Az alkalmazás kiolvassa az elsőt, és feldolgozza. Mindaddig, amíg ez a feldolgozás tart, a második üzenet nyugodtan várakozik a sorban. Ez azt jelenti, hogy ELVILEG a *Mégsem* gomb hatása csak a telepítés befejezése után lesz érezhető. Akkor pedig már semmit sem ér. Gyakorlatban viszont tudjuk, hogy ez nem így van. Van tehát megoldás a problémára, de vajon mi az?

Trükk: a telepítési folyamatot időnként (pl. egy-egy állomány átmásolása után) meg kell szakítanunk „egy gyors kitekintésre a nagyvilágba". Ez egy Windows függvény meghívásából áll (*ProcessMessages*). Ennek hatására programunk „kikukucskál" az üzenetsorába, és ha ott várakozó üzeneteket talál, akkor azokat feldolgozza. Az üzenetsor kiürítése után folytatódhat a telepítés. Ily módon még időben észre fogjuk venni, és fel is fogjuk dolgozni a *Mégsem* gombra való kattintás üzenetét.

1.2.4 Multitasking a 16 és 32 bites Windows verziókban

Vegyünk most egy más példát: mi történik, ha a telepítés alatt, a folyamat megszakítása nélkül, át szeretnénk lépni egy másik, eddig a háttérben futó alkalmazásba (Pl. a szövegszerkesztőbe). Az *Alt+Tab* billentyűkombináció hatására az aktuális alkalmazás kap egy inaktiváló, a szövegszerkesztő pedig egy aktiváló üzenetet (gyakorlatilag a két üzenet ugyanaz (*WM_ACTIVATE*), csak a paramétereikben van különbség). Ezen üzenetek feldolgozása különböző lesz a 16 és a 32 bites Windows verziókban.

- A 16 bites Windows változatokban a párhuzamosan futó alkalmazások között úgynevezett ***„non-preemptive multitasking"***-ot észlelünk. Ez azt jelenti, hogy egy alkalmazás egy adott üzenet feldolgozásának ideje alatt teljes mértékben uralja a rendszert. A vele párhuzamosan futó alkalmazások mindaddig nem fogadhatják saját üzeneteiket, amíg az előző be nem fejezte elkezdett üzenetének feldolgozását. Térjünk vissza telepítéses feladatunkhoz: a telepítés már elindult, mi pedig át szeretnénk lépni a szövegszerkesztőbe. Az *Alt+Tab* billentyűkombináció hatására bekerül ugyan a Word üzenetsorába az átváltás üzenete, azonban a szövegszerkesztő ezt csak a telepítés befejeztével fogja észrevenni és feldolgozni. Természetesen itt is alkalmazhatjuk a fent

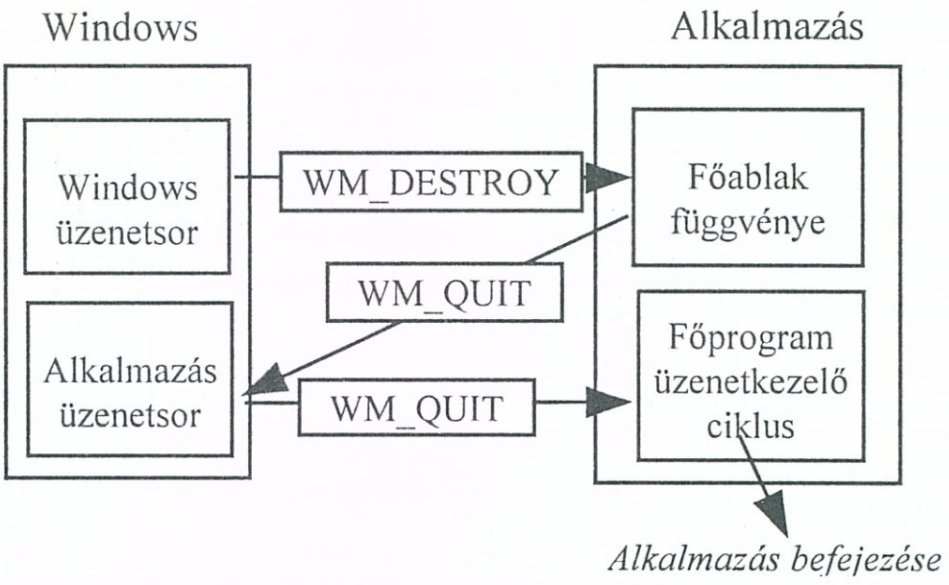
leírt trükköt, így az átváltás is sikeres lesz.

- A 32 bites Windows változatokban már „***preemptive multitasking***" van, azaz minden szál (*thread*) csak egy adott időfoszlány (időszelet) idejére „kapja meg a szót". (Senki nem lophatja el a processzort korlátlan ideig.) Az időfoszlány lejártával akarva-akaratlanul át kell adnia a szót a soron következő szálnak. Rövidesen újból rá fog kerülni a sor, így hát folytathatja az előbb abbahagyott tevékenységeket. Ha tehát telepítés közben lenyomjuk az *Alt+Tab* billentyűkombinációt, akkor a szövegszerkesztő üzenetsorába bekerül a *WM_ACTIVATE* üzenet, és az operációs rendszer jóvoltából a szövegszerkesztőnek lesz is alkalma ezt feldolgozni. Így hát minden trükk bevetése nélkül át tudunk váltani más alkalmazásokba.

A *preemtive multitasking* a párhuzamosan futó szálak között „tesz igazságot". Egy szálon belül a Windows 32 bites változataiban is az előbb bekövetkezett üzenet élvez elsőbbséget. A *Mégsem* gomb tehát itt is csak akkor fog érvényesülni, ha bevetjük a feljebb leírt trükköt.

1.2.5 Nem sorolt üzenetek

Vannak „nem sorolt" üzenetek is, azaz olyan üzenetek, melyeket a rendszer egyből az ablakfüggvényhez küld megkerülve az alkalmazás (szál) üzenetsorát. Ezek általában az ablakokat érintő üzenetek, mint például egy ablak létrehozása, bezárása. Ha például bezárjuk egy alkalmazás főablakát, akkor a *WM_DESTROY* üzenetet a rendszer egyenesen a főablak függvényének továbbítja (anélkül, hogy ez az üzenetkezelő cikluson átmenne). Ennek hatására az ablak bezárul, eltűnik, előtte viszont még elhelyez egy *WM_QUIT* üzenetet az üzenetsorában. Innen ezt az üzenetkezelő ciklus kiolvassa, és mivel ez pont a ciklus befejezésének feltétele, maga a főprogram is véget ér. Tehát előbb az ablak kapott egy jelzést (hogy tűnjön el), és csak utána lett vége a programnak is. Ha az üzenet feldolgozása a hagyományos módon zajlott volna le, akkor előbb a főprogram kapta volna meg az alkalmazás végét jelző üzenetet, így ennek ugyan vége szakadt volna, de az ablakai nem tűntek volna el!

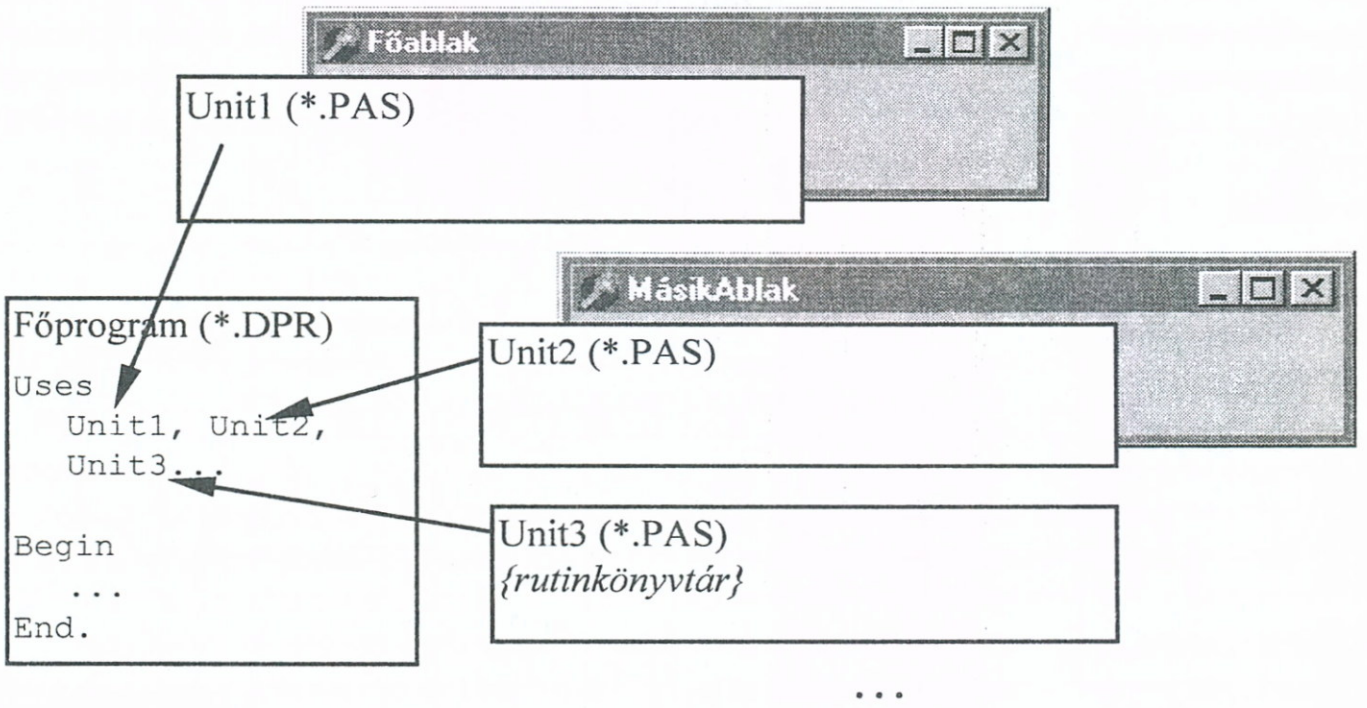


1.7. ábra. „Nem sorolt" üzenetek feldolgozása (*WM_DESTROY*)

2. Delphi bevezetés

Ebben a fejezetben megismerkedünk az általános Delphi alkalmazások szerkezetével, majd elkészítjük első alkalmazásunkat.

2.1 A Delphi alkalmazások felépítése



2.1. ábra. A Delphi alkalmazás felépítése

Minden Delphiben fejlesztett alkalmazásban megtalálhatók a következők:

- **Projektállomány** (***.DPR** = ***Delphi Project***)

  A Delphi alkalmazások főprogramját projektállománynak nevezzük, de szerepe megegyezik a hagyományos Turbo Pascal főprogram szerepével.

- **Űrlaptervek** (***.DFM** = ***Delphi Form***) és a viselkedésüket leíró **egységek** (***.PAS**)

  Egy Windows alkalmazásnak egy vagy több ablaka van. A Delphi egy vizuális fejlesztő eszköz, ami azt jelenti, hogy az ablakokat (űrlapokat, angolul *form*) vizuális

módon tervezzük meg. Már tervezés közben látható az ablak, elhelyezhetünk rajta akárhány szerkesztődobozt, gombot stb., méretezhetjük, mozgathatjuk ezeket. Az ily módon megrajzolt űrlapot a rendszer bináris formában tárolja egy DFM kiterjesztésű állományban. Természetesen minden ablaktervet külön állományban helyez el.

Windowsban az ablak interfész szerepet játszik a felhasználó és az alkalmazás között. A felhasználó az ablak elemeire hatva (menüpontok, gombok, választógomb-csoportok stb.) indítja el az alkalmazás különböző funkcióit. Ebből következik az, hogy az ablak maga és a rajta levő elemekkel indítható funkciók szorosan összefüggnek. Ezt az összefüggést a Delphi rendszer fejlesztői a következőképpen valósították meg: minden ablak külalakja egy DFM kiterjesztésű állományban, viselkedése pedig egy azonos nevű, de PAS kiterjesztésű állományban kerül tárolásra. A rendszer a közös név alapján egyértelműen el tudja dönteni minden ablakról, hogy hogyan néz ki, és hogyan viselkedik. Fordításkor (*compile*) a PAS egységekből **DCU** (***Delphi Compiled Unit***) állományok képződnek, programszerkesztéskor (*linking*) pedig a DFM és DCU párok az EXE részeivé válnak (2.2. ábra).

A Delphi egységek nyelve az Object Pascal, a Turbo Pascal továbbfejlesztett változata. E nyelv sajátosságait a 3. fejezetben ismerhetjük meg.

- **Rutinkönyvtárak** (opcionális)

  Ezek a következők lehetnek:

  - (Az ablakok viselkedését leíró egységeken túl) olyan **egységek (*.PAS)**, melyeknek rutin- és adatkönyvtár szerepük van, akárcsak a TP programokban.
  - Saját készítésű **DLL-ek**, melyek rutinjait szintén felhasználhatjuk alkalmazásunkban (adatait közvetlenül nem[1]!!!). Ezek nem szerkesztődnek hozzá programunkhoz, a rutinok hívása dinamikusan, futás közben történik.

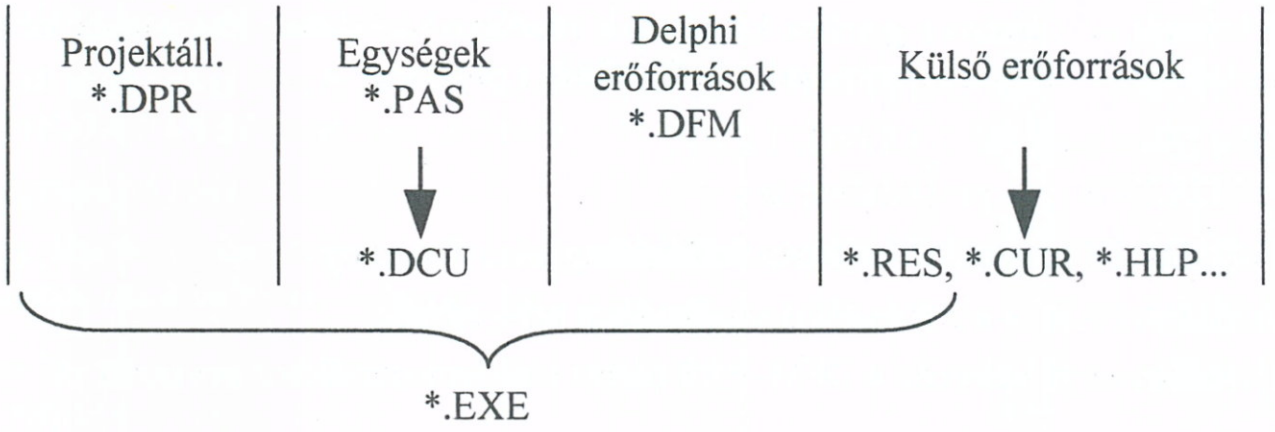
- **Külső erőforrások (*Resources*: *.ICO, *.CUR, *.ANI, *.BMP, *.RES, *.HLP stb.)**

  Alkalmazásunk minimizált képe a hozzárendelt *.ICO állományban tárolt képtől függ. Ha speciális – netán saját rajzolású – kurzorformákkal (*.CUR, *.ANI) vagy képekkel (*.BMP) szeretnénk dolgozni, akkor ezeket is alkalmazásunkhoz kell rendelnünk. Ezeket az erőforrásokat egy közös RES (***Resource***) kiterjesztésű állományban is elhelyezhetjük (ezt a Delphihez tartozó Image Editor segédprogrammal tehetjük). Az alkalmazásunk szerkesztésekor (*linking*) a RES tartalmát beépíthetjük az EXE állományba. (Saját rajzolású kurzorokkal a 4. fejezetben foglalkozunk.)

  Ha azt szeretnénk, hogy alkalmazásunk súgóval is rendelkezzen, akkor a szöveget előbb megszerkesztjük, standard súgóállomány formára hozzuk (fordítás útján → *.HLP), majd hozzárendeljük alkalmazásunkhoz. A súgóállományok készítésének módját a 16. fejezetben mutatjuk be.

---

[1] A DLL-ek rendelkeznek ugyan saját adatszegmenssel, de adataikra nem tudunk közvetlenül hivatkozni. Ezeket csakis interfész rutinok segítségével érhetjük el. Ugyanakkor a DLL a hívó alkalmazás vermét használja.



2.2. ábra. A Delphi alkalmazás felépítése

Mindezekből az alkotóelemekből (az esetleges DLL-ek, HLP-k és egyéb – az EXE-hez hozzá nem szerkesztett – erőforrások kivételével) fordítás és programszerkesztés útján létrejön a futtatható EXE állomány (2.2. ábra). Eltérően sok más windowsos alkalmazásfejlesztőtől, **a Delphi önálló futtatható állományt generál**. Egy nem adatfeldolgozó alkalmazás telepítésekor elég az EXE állományt, valamint a program által használt saját „extra" állományainkat (DLL-jeinket, HLP-jeinket...) vinnünk. Egy közepes méretű alkalmazás nem használ túl sok extrát (esetleg súgója van), így legtöbbször elegendő csak az EXE állományt telepíteni a célgépre, nem kell a különféle járulékos DLL állományokkal törődnünk. Az adatbázisos alkalmazásoknak bizonyos adatelérést megvalósító DLL állományokra is szükségük van. Bővebben lásd a 7. és 17. fejezetben.

Látható, hogy egy Delphi alkalmazás forráskódja több, különböző állományban kerül tárolásra. Fordítás során további állományok születnek. Minden Delphi alkalmazást külön könyvtárban helyezzen el!

Elemezzük az alkalmazást alkotó állományok szerkezetét egy egyszerű példaprogramon keresztül.

2.3. ábra. Első Delphi alkalmazásunk

Példaprogramunknak egy ablaka van (címe *PéldaAblak*), rajta egy szerkesztődoboz és két gomb látható: az *Üdvözlés* gombra kattintva a szerkesztődobozban megjelenik az ábrán látható szöveg, a *Kilépés* gombra pedig befejeződik a program.

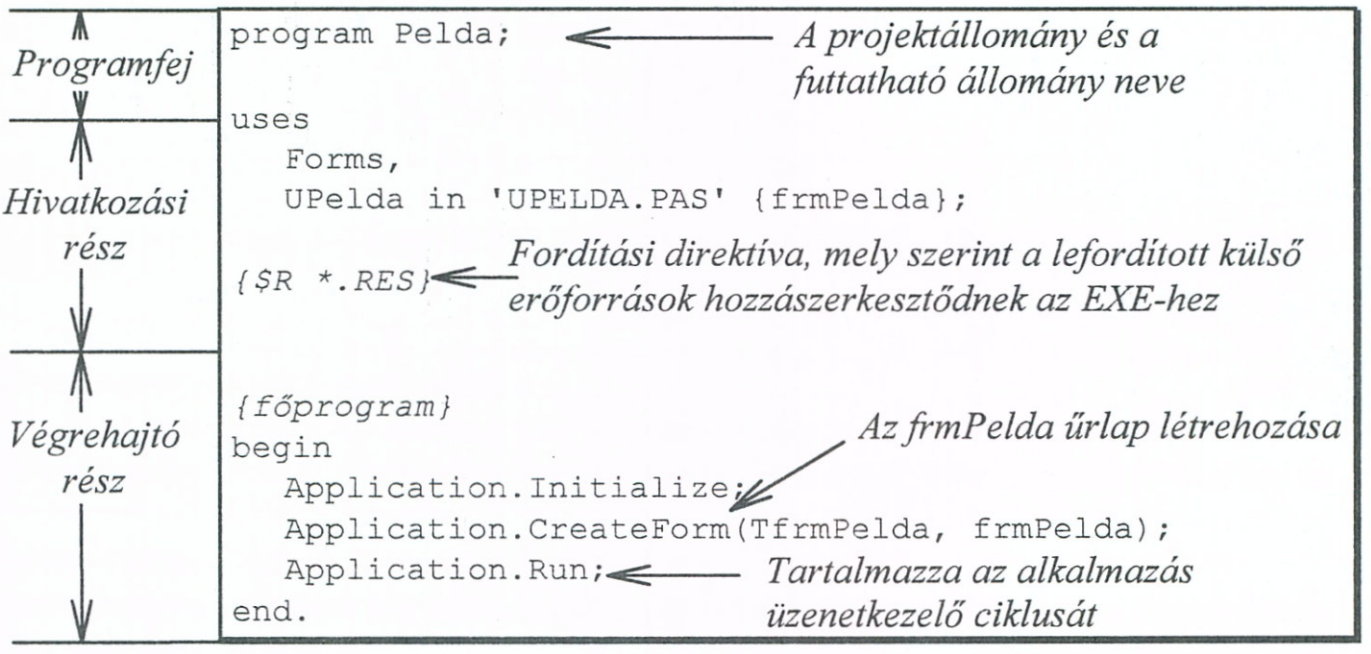
A feladat állományai (🖫 2_UDVOZLES\):

- PELDA.DPR – projektállomány
- UPELDA.DFM – a *PéldaAblakot* külalakilag leíró állomány
- UPELDA.PAS – a *PéldaAblak* viselkedését leíró egység

A projektállományt, valamint a DFM–PAS párost első lementésükkor nevezzük el.

2.1.1 A projektállomány szerkezete (*.DPR)

Ha egy új alkalmazást szeretnénk készíteni Delphiben, akkor meg kell hívnunk a *File/New Application* (Delphi 1-ben *File/New Project*) menüpontot. Ekkor a keretrendszer létrehoz egy projektállományt, egy űrlapot és az ehhez tartozó egységet. Ez azért van, mert az alkalmazásunkban egészen biztosan lesz egy főprogram és legalább egy űrlap, melynek a viselkedését is le kell valahol írni. Egy DPR, PAS és DFM állományra tehát minden alkalmazásban szükség lesz.



2.4. ábra. A projektállomány szerkezete (PELDA.DPR)

A projektállomány három fő részből áll:

- **Programfej:** semmiben sem különbözik a Pascalban megszokottól
- **Hivatkozási rész**

  Ez tartalmazza az alkalmazás által használt beépített (standard) egységek, valamint a saját egységeink és a hozzájuk tartozó űrlapállományok listáját. A mi esetünkben ez a *Forms* nevű Delphi standard egységet és a saját ablakunkat leíró egységet jelenti.

```
{EgységNév in 'EgységÁllomány' {ŰrlapNév}}
UPelda in 'UPELDA.PAS' {frmPelda};
```

Saját ablakunkat *frmPelda*-nak neveztük el, az őt leíró egység neve UPELDA, és az UPELDA.PAS állományban található.

- **Végrehajtó rész**

Egy általános objektumorientált alkalmazás főprogramja az alkalmazást inicializálja, futtatja, majd befejezi. Valahogy így:

```
Var
  Application: TApplication;
Begin
  Application.Init;  {inicializálás}
  Application.Run;   {futás}
  Application.Done;  {befejezés}
End;
```

Az *Application* a vezérlő objektum, osztálya a *TApplication*. Delphi alkalmazásunk főprogramja ehhez nagyon hasonlít (2.4. ábra), de vajon itt milyen szerepe van az *Application* objektumnak?

Minden windowsos alkalmazásban – mint ahogyan az előbbi fejezetben ezt láttuk – a főprogramnak azonos formája és szerepe van: tartalmaznia kell egy inicializáló részt (az ablakok létrehozását, megjelenítését), és tartalmaznia kell az üzenetkezelő ciklust is. Teljesen mindegy, hogy egy rajzoló-, egy szövegszerkesztő- vagy egy könyvelőprogramról van szó, a főprogram egészen biztosan ugyanígy fog kinézni. Ez a közös forma ihlette a Delphi keretrendszer szerzőit egy *TApplication* osztály megtervezésére (melyet a *Forms* egységben helyeztek el). Az így elkészült osztály summás leírása a következő:

- **Feladata:** egy általános windowsos alkalmazás adatainak tárolása, valamint feladatainak elvégzése
- **Adatok:** alkalmazásszintű információk, mint a futtatható állomány neve, ikonja, súgóállománya, főablaka[1] stb.
- **Metódusok**:

Initialize	Alkalmazás inicializálása. Az *Initialize* metódushívás csak a Delphi 32 bites verzióiban létezik, és itt is csak OLE automatizmus esetén van szerepe (bővebben lásd a 18. fejezetben).
CreateForm	A paraméterként átvett típussal és névvel rendelkező űrlap létrehozása

[1] A legtöbb windowsos alkalmazás több ablakkal rendelkezik, melyek közül egynek főablak szerepe van. Ez jelenik meg az alkalmazás elindításakor, és róla lehet majd később esetleges további ablakokat megnyitni. A főűrlap bezárása maga után vonja a program befejezését is.

Run Alkalmazás futtatása; ez tartalmazza az üzenetkezelő ciklust, mely minden windowsos alkalmazásban azonos.

...

Az *Application* nevű objektum deklarálása és inicializálása megtalálható a *Forms* egységben, így a főprogramban már csak az űrlapokat kell létrehoznunk (*CreateForm*), és máris jöhet az üzenetkezelő ciklus (*Run*). Ebben a pillanatban felbukkan a képernyőn alkalmazásunk főablaka, jelezvén, hogy készen állunk a felhasználó lépéseire: a menüpontok meghívása, a gombokra történő kattintás és általában minden felhasználó általi kezdeményezés üzenetek formájában alkalmazásunkhoz kerül. Ezeket programunkban – az üzenetkezelő ciklus jóvoltából – megfelelően fel is dolgozzuk. A ciklusból az alkalmazás befejezésekor lépünk csak ki, ekkor felszabadítjuk a lefoglalt memóriát (ez is a *Run* része), és ezzel vége a programnak.

Láthatjuk, hogy a projektállomány nem tartalmaz feladatspecifikus utasításokat. Elképzelhető, hogy egy rajzoló és egy szövegszerkesztő alkalmazásnak azonos tartalmú a főprogramja. Ez azért van, mert a főprogram csupán a főűrlap megjelenítéséért és a hozzá érkező üzenetek továbbításáért felelős. Az alkalmazás ablakainak konkrét kinézete és az elérhető funkcióiknak halmaza további állományoktól függnek (*.DFM és *.PAS). Ezek fogják a feladatspecifikus funkciókat ellátni.

A főprogramot csak nagyon indokolt esetben szerkesszük át. A teljes projektállományt a keretrendszer hozza létre, és ő is tartja karban. Ha további űrlapokkal bővítjük alkalmazásunkat, akkor a rendszer a főprogramban automatikusan el fogja helyezni az új ablakok hivatkozásait (*uses UÚjAblak...*).

2.1.2 Az űrlapállomány szerkezete (*.DFM)

Alkalmazásunk minden egyes űrlapjához tartozik egy-egy DFM állomány, melyben a rendszer az adott űrlap grafikus (külalaki) tulajdonságait tárolja bináris formátumban. Természetesen ez így számunkra használhatatlan, azonban át lehet alakítani szövegesre[1]. Az eredmény megtekinthető a 2.5. ábrán.

Hogyan jön létre ez az állomány? Egy új alkalmazás létrehozásakor megjelenik a képernyőn egy üres, legtöbbször szürke hátterű ablak. Ez alkalmazásunk egy ablaka, melyet nekünk kell megterveznünk. Megadhatjuk a címét, állíthatjuk ennek betűtípusát, stílusát. Ide-oda mozgathatjuk a képernyőn, ezzel befolyásolva a futáskori megjelenítésének helyét. Minden vizuálisan végrehajtott műveletünk eredménye rögtön lementődik ebbe az állományba. Mi több, ha konkrét feladatunkban egy szerkesztődobozra és két gombra van szükségünk, akkor ezeket is elhelyezhetjük az ablakon. Az új elemek vizuális tulajdonságai is azonnal bekerülnek az űrlapleíró állományba: pozíciójuk (a bal-felső sarkuk ablakrelatív koordinátái), méretük, esetleg a feliratuk (gomb esetén).

---

[1] A *.DFM → *.TXT konverziót a CONVERT.EXE program végzi el. Pl. CONVERT UPELDA.DFM. A Delphi 32 bites változataiban ugyanezt az űrlap gyorsmenüjében található *View As Text...* menüponttal is megtehetjük.

```
object frmPelda: TfrmPelda
   Left = 310
   Top = 18
   Width = 310
   Height = 130
   Caption = 'PéldaAblak'
   Font.Charset = EASTEUROPE_CHARSET
   Font.Color = clWindowText
   Font.Height = -13
   Font.Name = 'ARIAL CE'
   Font.Style = []
   PixelsPerInch = 96
   TextHeight = 16
   object eSzoveg: TEdit
      Left = 16
      Top = 24
      Width = 265
      Height = 24
      Font.Charset = EASTEUROPE_CHARSET
      Font.Color = clWindowText
      Font.Height = -13
      Font.Name = 'ARIAL'
      Font.Style = [fsBold]
      ParentFont = False
      TabOrder = 0
      Text = 'Hello! Ez már egy Delphi alkalmazás.'
   end
   object btnUdv: TButton
      Left = 48
      Top = 64
      Width = 75
      Height = 25
      Caption = 'Üdvözlés'
      TabOrder = 1
      OnClick = btnUdvClick
   end
   object btnKilep: TButton
      Left = 160
      Top = 64
      Width = 75
      Height = 25
      Caption = 'Kilépés'
      TabOrder = 2
      OnClick = btnKilepClick
   end
end
```

frmPelda adatok

eSzoveg adatok

btnUdv adatok

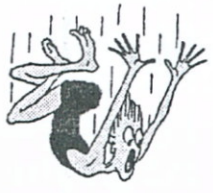
btnKilep adatok

2.5 ábra. Az UPELDA.DFM szövegesre visszafejtett leírása

Talán külön említést érdemel a *TabOrder* tulajdonság. Windowsban az űrlapokon található vezérlőelemek között – általános szokás szerint – a *Tab* billentyűvel lehet lépegetni előre és a *Shift+Tab*-bal visszafele. A bejárási sorrend az ablak tervezésekor dől el, mégpedig a *TabOrder* tulajdonság segítségével. Példaprogramunk elindításakor a szerkesztődoboz lesz fókuszban, hiszen ennek *TabOrder* értéke a legkisebb (0); *Tab* billentyűvel előbb az *Üdvözlés* feliratú gombra, majd a *Kilépés* feliratúra léphetünk, utána újra a szerkesztődobozba stb. Ha azt szeretnénk, hogy a szerkesztődoboz kimaradjon a láncból, csak a két gomb között tudjunk váltogatni, akkor az *eSzoveg.TabStop* tulajdonságot *False*-ra állíthatnánk. Ekkor már teljesen mindegy, mit állítottunk a *TabOrder*-be, a szerkesztődobozt bejáráskor átugorjuk.
Van itt még egy érdekesség: az *OnClick* tulajdonság. Egy gomb *OnClick* tulajdonságába kell beírnunk annak az eljárásnak a nevét, melynek a gombra való kattintáskor végre kell hajtódnia. Az eljárás kifejtése (teste) viszont már nem az űrlapleíró állományban kap helyet, hanem az űrlap viselkedését leíró egységben (lásd a következő pontban).

2.1.3 Az űrlaphoz tartozó egység (*.PAS)

A Delphi- és a TP-beli egységeknek azonos szerkezetük van: egységfej, illesztő, kifejtő és végrehajtó rész (2.6. ábra). Mivel itt egy űrlaphoz tartozó egységről van szó, ennek bizonyos részei űrlapspecifikusak.
Vegyük észre, hogy az általunk megtervezett űrlap számára a rendszer a háttérben egy vadonatúj osztályt hozott létre. Megvan benne minden, aminek egy általános ablakban lennie kell: címe, méretezhető kerete, rendszermenüje stb., létre lehet hozni, be lehet zárni. Mindezeket a *Forms* egységben leírt *TForm* osztálytól örökli (2.7. ábra). Űrlapunkat egy szerkesztődobozzal és két gombbal „gazdagítottuk", ezeket az új űrlaposztálynak tartalmaznia kell.
A szerkesztődoboz osztálya *TEdit*, a két gombé pedig *TButton*. Ezek mind létező, a Delphi keretrendszer által felkínált komponensek a *StdCtrls* egységből (bemutatásukat lásd 4. fejezetben). A *TfrmPelda* osztály tartalmazási kapcsolatban áll a *TEdit* és *TButton* osztályokkal. Két új metódust is definiáltunk, ezek – mint már ezt az űrlapállományban láttuk – a gombokra való kattintáskor kerülnek végrehajtásra. Az egész egységből csak a két új eljárás kifejtését kellett begépelnünk, minden egyebet a rendszer hozott létre vizuális tervezésünkkel párhuzamosan.



OOP-s szemszögből vizsgálva a két új metódust, felmerülhet a következő kérdés: ha a *btnUdvClick* a *btnUdv:TButton* gomb kattintására hajtódik végre, akkor ez miért az űrlaposztályban (*TfrmPelda*) található, és nem pedig a gombosztályban? Valóban az OO szemlélet azt sugallná, hogy a *btnUdvClick* metódust a gombosztályban implementáljuk, ekkor viszont a *btnUdv* gomb számára létre kellene hoznunk egy új, *TButton*-ból származó osztályt. Egy alkalmazásban azonban rengeteg gomb van, és ha mindegyikük számára új gombosztályt készítenénk, akkor igaz ugyan, hogy OO szemszögből támadhatatlan lenne alkalmazásunk, de ezért túl nagy árat fizetnénk.
Ehelyett Delphiben az új metódusokat az űrlaposztályban hozzuk létre, és csupán egy jellemző — a *TButton.OnClick* — árulja el, hogy az új metódus tulajdonképpen a gombhoz tartozik.

Egységfej

Illesztő rész

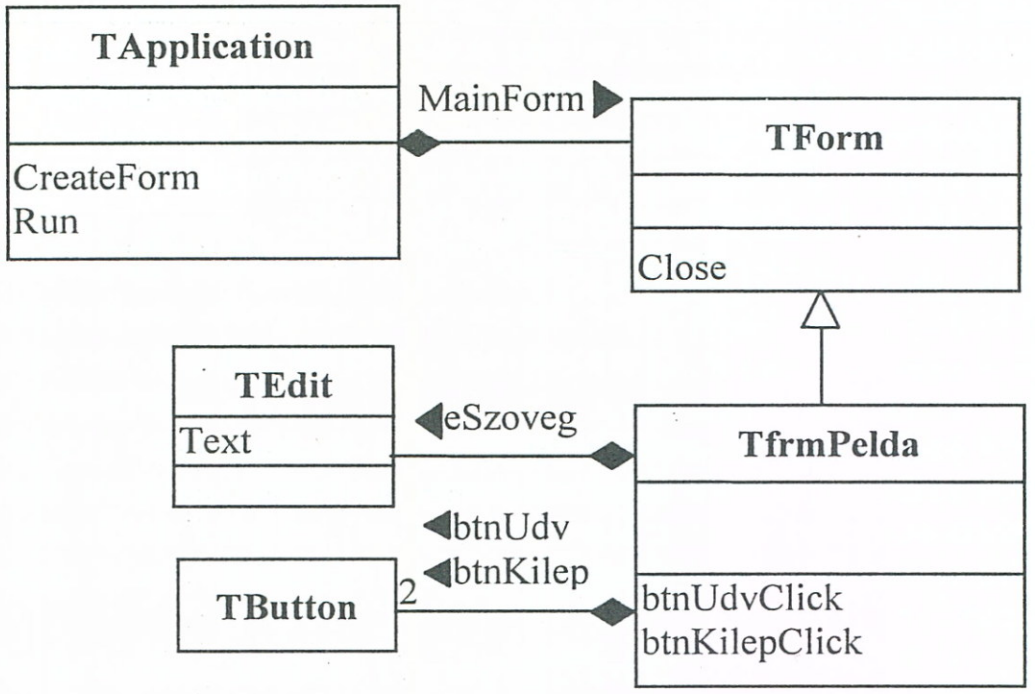
Kifejtő rész

Inicializációs rész (üres)

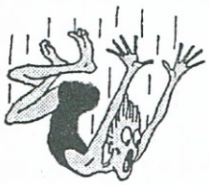
```
unit UPelda;
                    Automatikusan beépített Delphi könyvtári egységek
interface
uses Windows, Messages, SysUtils, Classes, Graphics,
     Controls, Forms, Dialogs, StdCtrls;
type                                  TfrmPelda osztály deklarációja
    TfrmPelda = class(TForm)
       eSzoveg: TEdit;
       btnUdv: TButton;
       btnKilep: TButton;
       procedure btnUdvClick(Sender: TObject);
       procedure btnKilepClick(Sender: TObject);
    private
       { Private declarations }
    public
       { Public declarations }
    end;
                                  frmPelda objektum deklarációja
var frmPelda: TfrmPelda;
implementation                    TfrmPelda osztály kifejtése
{$R *.DFM}
procedure TfrmPelda.btnUdvClick(Sender:TObject);
begin
  eSzoveg.Text := 'Hello! Ez már egy Delphi alkalmazás.';
end;

procedure TfrmPelda.btnKilepClick(Sender: TObject);
begin
  Close;
end;

end.
```

2.6. ábra. Az űrlaphoz tartozó egység szerkezete

De vajon hogyan kapcsolódik a mi ablakunk az alkalmazáshoz? Az alkalmazást az *Application* objektum képviseli: ennek van egy *MainForm* nevű mezője, mely az alkalmazás főablakát tartalmazza. Ebben a feladatban csak egy ablak van, az *frmPelda*, ez egyben főablak is. Az űrlapobjektumra két programmodulban is hivatkozunk: leírása és deklarációja az űrlap egységében található (*var frmPelda:TfrmPelda...*), létrehozása pedig a főprogramban (*CreateForm(TfrmPelda, frmPelda)*). Emiatt van szükség a főprogramban a *Uses UPelda* beszerkesztésére (ezt már a 2.4. ábrán láttuk).



2.7. ábra. Alkalmazásunk osztálydiagramja (UML[1] jelölést alkalmazva)



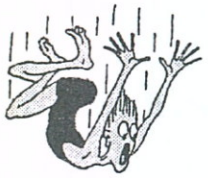
OK, mondhatja a kedves Olvasó, ez mind szép és jó, de hogyan működik ez az alkalmazás? Honnan tudja a rendszer, hogy amikor az *Üdvözlés* gombra kattintunk, akkor az *frmPelda BtnUdvClick* metódusát hívja meg?
A válasz nagyon egyszerű. Azt már láttuk, hogy az üzenetkezelő ciklus a főprogramban – az *Application.Run* metódushívásban – található. Ez sorban kiolvassa, majd továbbítja az alkalmazás üzeneteit a megfelelő ablakfüggvénynek. De hova rejtették el az ablakfüggvényeket? E kérdés megválaszolásának céljából tekintsük át egy pillanatra a Delphi osztályhierarchiáját (2.8. ábra).

A rendszer felkínál számos komponenst, úgy egyed, mint interfész és kontroll jellegűeket[2]. Egyed típusú például a *TDatabase* vagy a *TClipboard* objektum, interfész a *TButton*, és kontroll jellegű a *TApplication*. Már most gyaníthatjuk, hogy a Delphiben egyszerű lesz a rendszer-párbeszédablakok megjelenítése, menük létrehozása stb., hiszen mindezeket már osztályok (komponensek) formájában implementálva kapjuk. Elég tehát példányokat létrehoznunk, melyeket

---

[1] Programjaink osztálydiagramjainak elkészítésében az UML (**U**nified **M**odeling **L**anguage = Egyesített Modellező Nyelv) jelölést alkalmazzuk. Ez egy objektumorientált rendszerfejlesztési módszer, mely — amint a neve is mondja — három nagy módszer (Booch, OMT és OOSE) egyesítésével 1997-ben látott napvilágot, és azóta egyre nagyobb teret hódít az objektumorientált világban.

[2] Az objektumokat alapvetően három típusba sorolhatjuk:
Egyed: adatot tároló, valós objektum
Interfész: kapcsolatteremtő, megjelenítő objektum
Kontroll: vezérlést végző objektum



kényünk-kedvünk szerint testre szabhatunk. Az itt látható osztályok zömét megismerhetjük a következő két fejezetben. Másokat később mutatunk be, pl. a *TDatabase* osztályt a 8. fejezetben, a *TClipboard*-ot a 18. fejezetben.
De térjünk vissza örökzöld témánkhoz: hogyan fut az alkalmazás? Hol rejtőznek az ablakfüggvények?
A *TWinControl* osztályban megjelenik egy *MainWndProc* nevű metódus. Ezt a metódust minden leszármazottja örökli, tehát a *TButton* is. Ez a keresett ablakfüggvény. Ha az *btnUdv*-ra (*TButton* példányra) kattintunk, akkor a Windows az üzenetet a *MainWndProc* ablakfüggvénynek adja át. Ez feldolgozza, mégpedig úgy, ahogyan az egy gombtól elvárható: a gomb „benyomódik", végrehajtódik az *OnClick* eseményére írt metódus, majd „visszaugrik". És mivel mi a *btnUdvClick* metódust az *OnClick* eseményre építettük, ez fog lefutni, azaz megjelenik a szerkesztődobozban a kívánt szöveg: 'Hello! Ez már egy Delphi alkalmazás.' (És remélem, nem is egy akármilyen, hanem egy megértett alkalmazás!)

```
TObject                          {A Delphi osztályok közös őse}
  Exception                                  {Kivételosztályok}
    EInOutError
  TList                          {Általános konténerosztály}
  TPersistent
    TClipboard                                        {Vágólap}
    TComponent                               {Komponensek őse}
      TApplication               {Az alkalmazás osztálya}
      TCommonDialog                          {Párbeszédablakok}
        TOpenDialog
        TPrinterSetupDialog
      TControl
        TWinControl              {„Ablakozott" komponensek}
          TButtonControl                               {Gombok}
            TButton
              TBitBtn
          TCustomEdit                      {Szerkesztődoboz}
            TEdit
          TScrollingWinControl
            TForm                                      {Űrlapok}
              TfrmPelda          {Ez a mi űrlapunk!}
      TMenu                                            {Menü}
      TDatabase                          {Adatbáziskomponens}
    TStrings                     {Általános karakterlánclista}
```

2.8. ábra. „Ízelítő" a Delphi osztályhierarchiájából

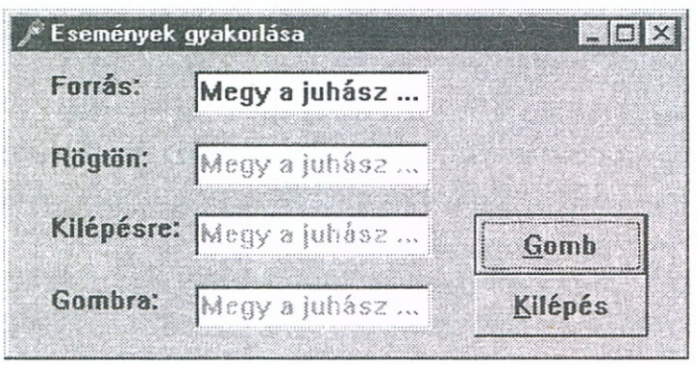
2.2 Egy egyszerű Delphi alkalmazás elkészítése

Egy egyszerű Delphi alkalmazás elkészítése a következő lépéseken keresztül történik:

- Az alkalmazás űrlapjainak, menüszerkezetének megtervezése, kitalálása
- Az űrlapok kivitelezése, megrajzolása
- Az egyes gombok, menüpontok... eseménykezelőinek megírása
- Az alkalmazás tesztelése

Bármilyen alkalmazáshoz tehát ismernünk kell a rendelkezésünkre álló komponenseket (szerkesztődoboz, gombok, választógombok...), ezek jellemzőit, eseményeit. Mindezekkel részletesen a 4. fejezet foglalkozik, de addig is – a keretrendszerrel való ismerkedésként – oldjunk meg együtt egy feladatot.

Feladat: Készítsünk egy alkalmazást, melynek ablakán négy szerkesztődoboz van: *Forrás*, *Rögtön*, *Kilépésre* és *Gombra*. Csak az első (*Forrás*) legyen szerkeszthető. A másik három szövege kövesse a *Forrás* szövegét: a *Rögtön* szerkesztődobozban azonnal jelenjen meg a szöveg, ahogyan azt a forrásba begépeljük. A *Kilépésre* dobozban akkor jelenjen meg a szöveg, amikor elhagyjuk a *Forrás* szerkesztődobozt (pl. *Tab* hatására más vezérlőelemre kerül a fókusz), a negyedik (*Gombra*) dobozban pedig egy gomb hatására történjen mindez. Lehessen az alkalmazást bezárni egy gomb segítségével.

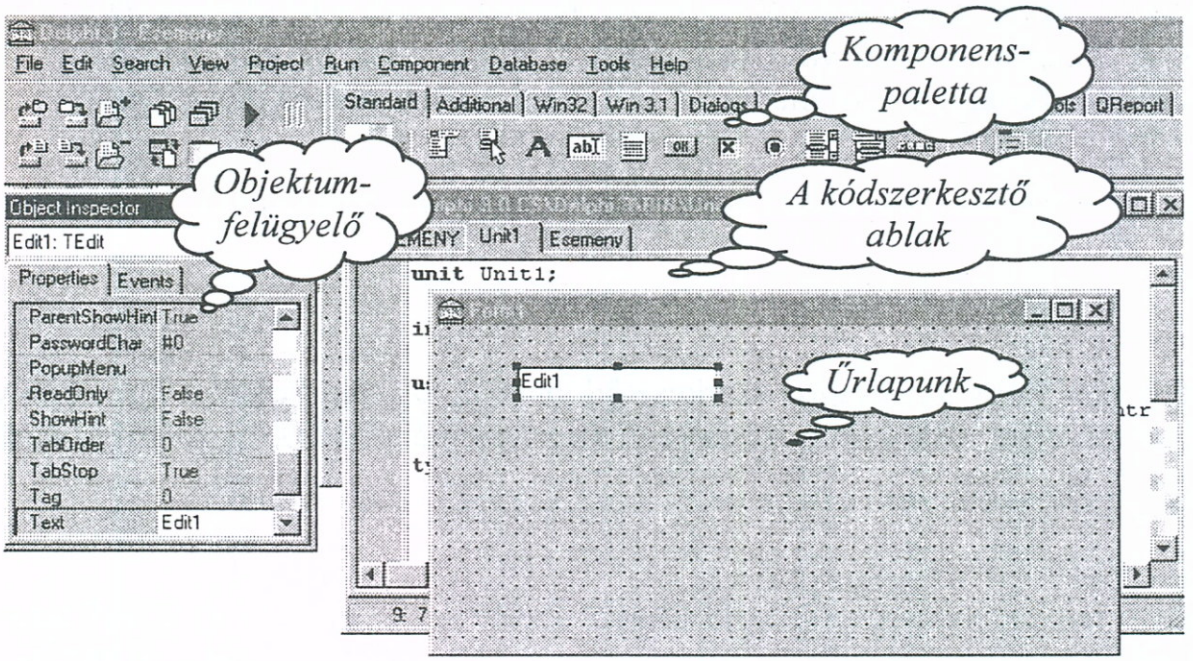


2.9. ábra. Események gyakorlása

Megoldás (🖫 2_ESEMENYEK\ESEMENY.DPR)

Kezdjük a feladat megoldását egy új alkalmazás létrehozásával. Hívjuk meg a *File/New Application* menüpontot (Delphi 1-ben *File/New Project*). Hatására a rendszer létrehoz egy üres űrlappal rendelkező alkalmazást (2.10. ábra).

Az űrlap megtervezése a komponensek „összecsipegetésével" kezdődik: a megfelelő komponenspalettáról megfogjuk a szükséges komponenst, majd ezt elhelyezzük az űrlapon. Az objektum-felügyelőben mindig a kijelölt űrlapelem tervezési időben (*design time*) elérhető jellemzőit (*properties*) és eseményeit (*events*) láthatjuk, állíthatjuk. Ilyen például a *Name* jellemző, mellyel minden komponens rendelkezik, és ezek programbeli nevét tartalmazza.



2.10. ábra. A keretrendszer főbb elemei

Tervezzük meg feladatunk űrlapját. Helyezzük el rajta a négy szerkesztődobozt (*TEdit*), a négy címkét (*TLabel*) és a két gombot (*TButton*). Mindezek a *Standard* komponens-palettán találhatók. A komponensek beállításait a következő táblázat mutatja:

Komponens	**Jellemző**	**Érték**
eForras[1]: TEdit	Text	" *{az elején még üres}*
eRogton, eKilepesre, eGombra: TEdit	Text	"
	Enabled	False *{nem szerkeszthető}*
btnGomb: TButton	Caption	&Gomb *{⇨Gomb és Alt+G forróbillentyű}*
btnKilepes: TButton	Caption	&Kilépés

A címkék általában csak információs szerepet játszanak, nem ők képezik a feladat lényegét. Emiatt a komponensek beállításait tartalmazó táblázatban ezeket nem tüntetjük fel. Példánkban állítsuk be a négy címke *Caption* tulajdonságát a 2.9. ábra szerinti szövegekre.

---

[1] Miután elhelyezünk űrlapunkon egy komponens-példányt, első dolgunk az, hogy átnevezzük, azaz a *Name* jellemzőjét beállítjuk egy szuggesztív névre. A feladat-megoldásokban a beállítandó név a táblázat első oszlopában lesz feltüntetve, pl. *eForras:TEdit*. Minden komponens nevében feltüntetünk egy előtagot, mely a típusára utal (***e**Forras* ⇨ ***Edit**Forras*), így a programban „első ránézésből" tudni fogjuk, milyen jellemzőkkel, metódusokkal rendelkezik.

Nézzük, mit kell tenni annak érdekében, hogy alkalmazásunk a feladatspecifikáció szerint működjön. A *Kilépés* gombra való kattintásra a programnak be kell fejeződnie. Erre a *TForm* osztálytól örökölt *Close* metódust kell meghívnunk a *btnKilepes.OnClick* eseményében[1]:

```
procedure TfrmEsemenyek.btnKilepesClick(Sender: TObject);
begin
  Close;
end;
```

Annak érdekében, hogy az *eRogton* szerkesztődobozban azonnal lássuk a forrásban végrehajtott változtatásokat, az *eForras* doboz *OnChange* eseményére a következőket írjuk[2]:

```
procedure TfrmEsemenyek.eForrasChange(Sender: TObject);
pegin
   eRogton.Text := eForras.Text;
end;
```

A *Kilépésre* doboz szövegét az *eForras*-ból való kilépéskor kell átírni ⇒ *eForras.OnExit*

```
procedure TfrmEsemenyek.eForrasExit(Sender: TObject);
begin
   eKilepesre.Text := eForras.Text;
end;
```

Az *eGombra* dobozban a szövegnek akkor kell megjelennie, amikor a *btnGomb*-ra kattintunk. Szövegének beállítását tehát a *btnGomb.OnClick* eseményére fogjuk építeni:

```
procedure TfrmEsemenyek.btnGombClick(Sender: TObject);
begin
   eGombra.Text := eForras.Text;
end;
```

Próbálja ki az alkalmazást! A gombok forróbillentyűk segítségével is meghívhatók (Alt + az aláhúzott betű).

Remélem a kedves Olvasó kedvet kapott a delphis alkalmazások készítéséhez. Sajnos még nem tudunk eleget ahhoz, hogy bonyolultabb feladatokat is megoldjunk. A következő két fejezet ezt a hiányt próbálja pótolni. A 3. fejezetben megismerkedhetünk az Object Pascal nyelv sajátosságaival az ismertnek tekintett Turbo Pascalhoz képest. A 4. fejezetben bemutatjuk a leggyakrabban használt komponensek fontosabb jellemzőit és eseményeit, mindezzel megalapozva a későbbi munkát.

---

[1] Ilyenkor az objektum-felügyelőben duplán kattintunk a *btnKilepes OnClick* esemény melletti „fehér dobozkában". Hatására megjelenik a kódszerkesztő ablak, melyben a metódus váza már le is lett generálva, nekünk csak a kifejtését kell megírnunk.

[2] Most az *eForras* szerkesztődoboz *OnChange* eseményénél kattintsunk duplán az objektumfelügyelőben.

3. A Turbo Pascaltól az Object Pascalig

Az Object Pascal (a későbbiekben OP) a Delphi objektumorientált nyelve. Ebben a fejezetben az OP és a már ismertnek tekintett Turbo Pascal (TP) közötti fontosabb különbségekről lesz szó. Ezt a fejezetet főképpen azoknak az Olvasóknak javaslom, akik már objektumorientált programozási szinten ismerik a TP-t, de a Delphivel még nem, vagy alig foglalkoztak. Itt elsajátíthatják azokat az ismereteket, melyek a Delphiben való programozáshoz nélkülözhetetlenek.

3.1 A különbségek és újdonságok rövid áttekintése

Ismerkedjünk meg az OP nyelvben bevezetett újdonságokkal a TP-hez képest. Tapasztalni fogjuk, hogy bizonyos elemek változatlanul használhatók itt is, mások kicsit megváltoztak, megint mások pedig teljesen értelmüket vesztették. Természetesen számos újdonság is van.

- **Mi marad a TP-ből?**
  - Program és egység váza
  - Vezérlőszerkezetek (szekvencia, szelekció, iteráció)
  - Típusok és rutinok nagy része

    A rutinkészlet kibővült, a régi TP rutinok zöme azonban továbbra is hívható. A 3.2. pontban megismerkedhetünk néhány új és nagyon hasznos rutinnal.
- **Mit nem használunk az OP-ban?**
  - A TP-beli billentyűzetről beolvasó és képernyőre kiíró eljárásokat és függvényeket. Ezekre a Delphiben nem lesz szükségünk, mivel itt egy adat bekérése és megjelenítése szerkesztődobozokban vagy más hasonló komponensekben történik a felhasználó kezdeményezésére. A szöveges, típusos stb. állományok kezelése viszont lényegében nem változott. (*Assign* helyett *AssignFile* van, *Close* helyett pedig *CloseFile*.)
- **Kicsit más, mint a TP-ben:**
  - Az osztálymodell. Részletes bemutatása a 3.3. pontban található.

- A memóriamodell. A Delphi 32 bites verzióiban megszűnik a lefoglalható memória maximum 64KB-os korlátozása, itt már lefoglalható a Windows teljes virtuális memóriaterülete (2GByte).

- **Újdonságok:**
  - OP-ben a függvényeknek összetett típusú visszatérési értékük is lehet (pl. rekord vagy tömb, lásd 3.4. pont).
  - Az Object Pascal programban lehetőség van a hibalogika és a normál programlogika teljes elkülönítésére. Megírjuk rutinunkat az általános esetre, nem figyelve minden utasításnál az esetleges hibalehetőségekre, majd a rutin végén felsoroljuk, hogy milyen hiba előfordulása esetén, hogyan reagáljon a program. Mindez a kivételek kezelésével valósul meg (lásd 3.5. pont).
  - A Delphi 1-ben megjelenik a *PChar* karakterlánc típus, a C sztring megfelelője, azaz a nulla-végű karakterlánc. Ezzel az új típussal kiküszöbölhetjük a TP-beli *String* típus hátrányait (a max. 255 karakteres korlátot), és nem utolsósorban kényelmesebben hívhatjuk az API rutinokat (ezek mind nulla-végű karakterláncot várnak paramétereikben). Ezen kívül a Delphi 32 bites változataiban további karakterlánc-típusok is megjelennek: a *ShortString*, az *AnsiString* és a *WideString*. Mindezekről részletesebben a 3.6. pontban lesz szó.
  - Létezik egy általános konténer osztály (egy lista), a *TList*. Ebbe bármit el lehet helyezni az egész számoktól, a rekordokon át egészen az objektumokig. Bővebben lásd a 6. fejezet 6.3.3. pontjában.
  - A Delphi 32 bites verzióiban megjelenik a *Variant* adattípus (sokan ezt Visual Basicből már ismerik). A *Variant* adatokra nem jellemző a típusuk, futáskor bármilyen értéket felvehetnek. A pascalos neveltetésű programozók szemében ez „maga az ördög", mégis vannak esetek, amikor használnunk kell. Ilyenek az adathalmazokban való keresést megvalósító *Locate* és *Lookup* függvények (8. fejezet), valamint az OLE automatizmus (18. fejezet).

3.2 Új, hasznos rutinok

Ebbe a kategóriába tartoznak a dátum- és időkezelő rutinok, valamint a karakterlánc és számok közötti konverziós függvények:

- Dátum- és időkezelés:

  Létezik egy *TDateTime* típus[1], mely valós formátumban tárolja a dátum és az időpont információt: egész részében a dátumot, tört részében pedig az időt.

  A dátum és idő kezelésére leggyakrabban használt rutinok:

  - *Date*: visszaadja a mai dátumot; *Time*: visszaadja a pontos időt

[1] A TP-ben is létezik *TDateTime* típus; az ott egy rekord, mely rendelkezik a *Year*, *Month*... mezőkkel.

- *DecodeDate(Datum, Ev, Ho, Nap)*: egy dátumból kiolvassa az év, hó és nap információkat; fordítottja az *EncodeDate(...)*
- *DecodeTime(Ido, Ora, Perc, MasodPerc, EzredMasodPerc)*: időérték lebontása; párja az *EncodeTime(...)*

- Konverziós függvények:

  Szám, dátum átalakítása karakterlánccá:

  - *IntToStr, FloatToStr, DateToStr, TimeToStr, DateTimeToStr*
    Például:

```
Var S:String;
...
S:= IntToStr(18); {S ← '18'}
S:= IntToStr(18.39); {S ← '18.39'}
```

  Karakterlánc átalakítása számmá, dátummá:

  - *StrToInt, StrToFloat, StrToDate, StrToTime, StrToDateTime*
    Például:

```
Var I:Integer;
...
{Az I változóba beállítjuk a szerkesztődoboz egésszé konvertált
értékét.}
I:= StrToInt(eSzerkDoboz.Text);
```

Egy karakterlánc számmá alakítása nem mindig sikeres. Ilyenkor a konverziós függvény egy *EConvertError* kivételt generál, lásd a 3.5. pontban.

3.3 Az Object Pascal osztálymodell

3.3.1 Az osztály deklarációja

TP7-ben egy osztályt az *Object* kulcsszó segítségével deklarálunk. OP-ben egy osztályt hagyományos módon is (*Object*) deklarálhatunk, de ha az újdonságokat ki akarjuk használni, akkor ajánlatos az új típust, a *Class*-t használni.

```
Type
  TOsztály = Class (TŐsOsztály)
    Mezőlista
    Metóduslista
    Jellemzőlista
  End
```

Azt már látjuk, hogy az osztály deklarációjában új elemek is megjelennek (jellemzőlista), ezek bemutatása előtt viszont tekintsünk át néhány osztályszintű újdonságot:

- A *Class*-ként deklarált osztály példányai dinamikus objektumok. Egy objektumot TOsztály-nak deklarálunk (*Var Obj:TOsztály*), mégis a fordító *Var Obj:^TOsztály* típusúnak veszi, azaz tulajdonképpen egy objektum-mutatónk lesz. Az objektum számára inicializáláskor foglalódik le a szükséges hely, **a konstruktor mindig a példány címével tér vissza**. Ez egy örvendetes dolog, hiszen így az objektumok csak inicializálásuk után foglalnak helyet a memóriában egészen a megszüntetésükig, tehát ezzel memóriaterületet spórolunk meg. Természetesen a dinamizmusnak más következményei is vannak: változik az objektumok inicializálásának módja. Hagyományos módon (*Obj.Create(...)*) nem történhet, mivel inicializálás előtt az objektum még nem jött létre, tehát még nincs helye a tárban, és emiatt nem hívhatjuk meg a metódusait, így a konstruktorát sem. Az új típusú objektum helyes életrekeltésének módja a következő:

```
Var
  Obj : TOsztály;
  ...
  Obj := TOsztály.Create (paraméterek)
  {Obj.Create(...)💣}
```

A dinamikus létrehozásnak egy másik lehetséges következménye az adatokra és metódusokra való nehézkesebb hivatkozás („kalapozás") lehetne. Ezt a problémát azonban megoldották, beépítették a fordítóprogramba. Az objektumok metódusaira, jellemzőire és – ha nagyon akarjuk – adataira is (habár tudjuk, hogy ezt közvetlenül „nem illik" megtenni), továbbra is a hagyományos módon hivatkozunk: *Obj.Metódus(paraméterek), Obj.Jellemző, Obj.Adat*. A fordítóprogram minden új típusú objektum esetén a hivatkozási láncba beilleszti a „kalapot", azaz a ^ jelet, és ezt nyugodtan teheti, hiszen minden objektum dinamikusan jön létre. A lényeg tehát az, hogy a hivatkozás cseppet sem lett nehezebb, az előnyök viszont érezhetőek.

- Egy másik újdonság a „közös ős" létezése. Az új osztálymodellben minden osztálynak – sem több, sem kevesebb, mint – pontosan egy őse van. Ez vagy az explicit módon feltüntetett ős (1. eset), vagy az implicit *TObject* osztály, ha ezt nem tüntetjük fel tételesen (2. eset).

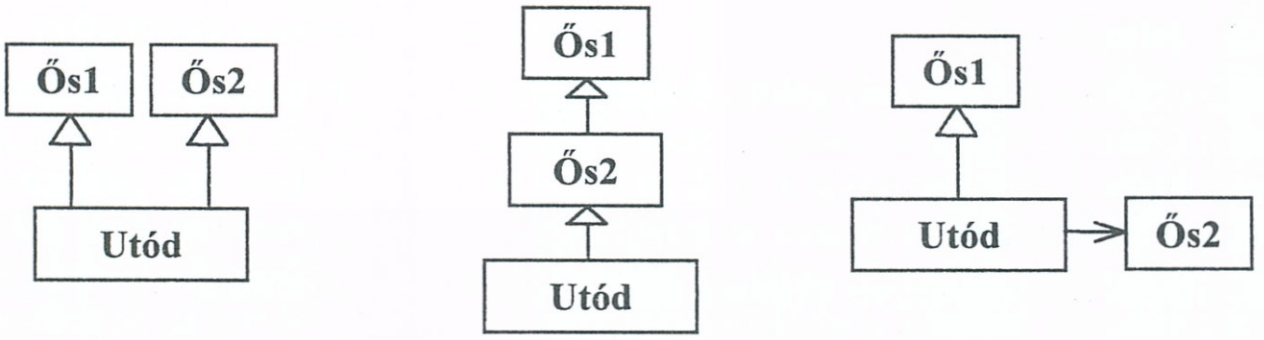
```
TOsztály = Class (TŐsOsztály)
   Mezőlista
   Metóduslista
   Jellemzőlista
End
```

1. eset: Explicit ős (TŐsOsztály)

```
TOsztály = Class
  Mezőlista
  Metóduslista
  Jellemzőlista
End
```

2. eset: Implicit ős (TObject)

Többszörös öröklés itt sem lehetséges, akárcsak a TP - ben, azonban mint tudjuk, ez mesterségesen két módszerrel is megvalósítható (lásd 3.1. ábra).



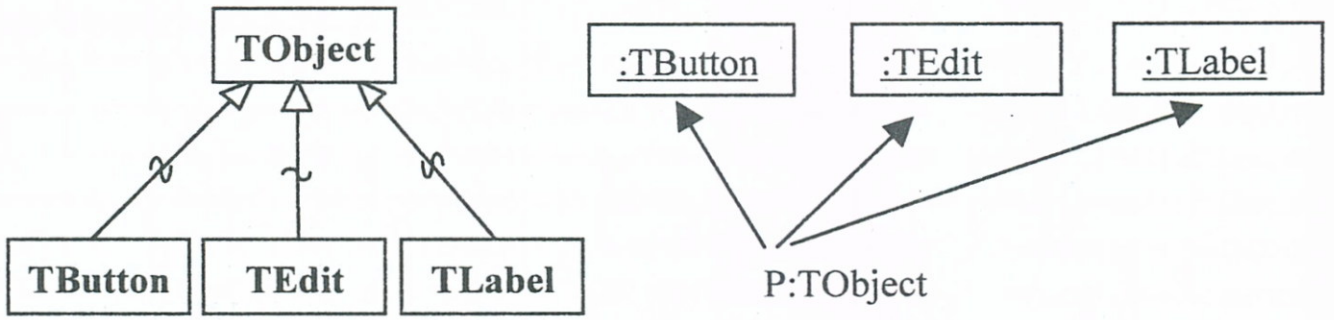
Ezt szeretnénk megvalósítani

Első megoldás: az egyik ős mesterséges befordítása a másik alá

Második megoldás: az egyik ős kapcsolatként való felvétele az utódban

3.1. ábra. Többszörös öröklés és megoldásai

Tehát az új osztálymodellben minden osztálynak van egy közös őse, a *TObject* osztály. De vajon miért jó ez? Gondoljuk át a következőket: egy adott űrlap nyilván tárolja valahogy az alkotóelemeit (gombjait, szerkesztődobozait, rendszer párbeszédablakait...), hiszen az űrlap bezárásakor az ő feladata az elemeinek eltüntetése. Valószínű, hogy egy listára fűzi fel ezeket. Igen ám, de hogyan lehet egy közös listára felfűzni különböző típusú elemeket? Ez csak akkor lehetséges, ha a felfűzendő elemek közös össel rendelkeznek, mivel egy közös ős típusú mutatóval később rámutathatunk a konkrét utódosztály típusú elemekre. A lista elemeit tehát *TObject*-nek deklaráljuk, és később a konkrét elemek akár *TButton*, *TEdit*... példányok is lehetnek. Ezt szemlélteti a 3.2. ábra is.



3.2. ábra. A közös ős előnye

A *TButton*, *TEdit*... osztályok közös (de nem közvetlen) őse a *TObject*. Ha *P* egy *TObject* típusú objektum-mutató (!), akkor vele mutathatunk bármilyen utód típusú objektumra. Tehát a lista elemeinek típusa lehet a *TObject*.

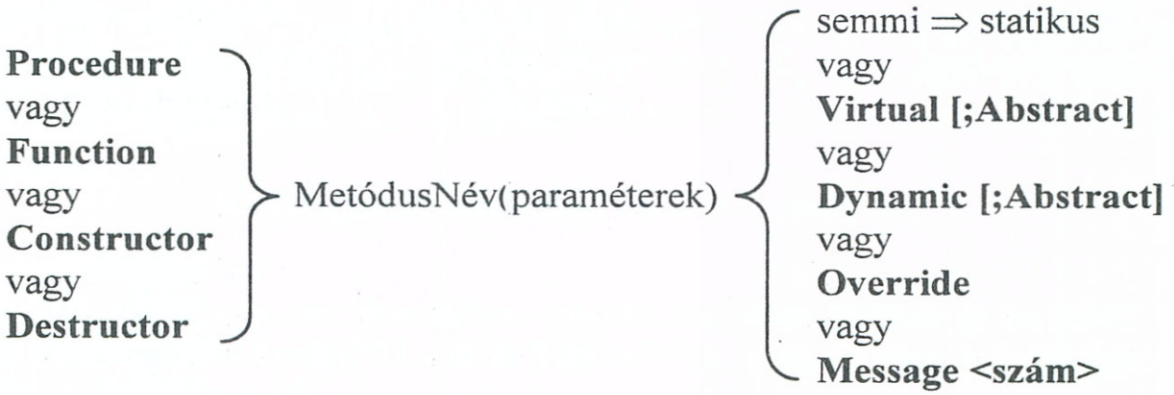
Most pedig vegyük sorba az osztály elemeit, ismerkedjünk meg velük részletesebben.

3.3.2 Mezőlista

Itt kell felsorolnunk az osztály változóit ugyanúgy, ahogyan ezt a TP-ben is tettük.

3.3.3 Metóduslista

A metódusok deklarációjának általános formáját a 3.3. ábra szemlélteti:



3.3. ábra. Metódusok

A metóduslista tartalmazza az osztályhoz tartozó rutinok (függvények és eljárások) deklarációit. Minden osztálynak van egy konstruktora (életrekeltő, inicializáló metódusa) és egy destruktora (megszüntető metódusa). TP-ben a konstruktort *Init*-nek, a destruktort *Done*-nak hívják, OP-ben viszont a következő a névkonvenció:

- Konstruktor: **Create(paraméterek)**
- Destruktor: **Destroy**{általában nincsenek paraméterei}

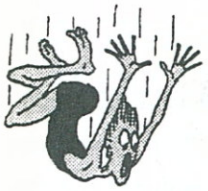
Mivel az új *class*-ként deklarált osztályok példányai dinamikusan jönnek létre, a destruktor feladata az objektum által lefoglalt memóriaterület felszabadítása is. Az űrlapok vizuális tervezésekor a rendszer automatikusan meghívja az űrlapon elhelyezett elemek konstruktorait, futási időben pedig, az űrlap bezárásakor, meghívja ezek destruktorait. Ha viszont mi programból szeretnénk létrehozni bizonyos objektumokat, akkor ezek megszüntetését is nekünk kell elvégeznünk. Így elvileg előfordulhat (természetesen tévedésből), hogy meg akarunk szüntetni egy nem inicializált objektumot. Mi történik ilyenkor? A még le sem foglalt memóriaterület fog felszabadulni?! Nyilvánvaló, programunk a destruktor hívásakor futási hibával leáll. Ahhoz, hogy ez ne fordulhasson elő, a *Destroy* helyett használjuk a ***Free*** metódust. Ez előbb leellenőrzi az objektum-mutatót, és csak akkor hívja meg a destruktort, ha ez nem *Nil*.

Statikus, virtuális és dinamikus metódusok

Értelmezzük a 3.3. ábra jobb felét! Ha semmit nem írunk egy metódus után, akkor az statikus[1] lesz. A virtuális és dinamikus metódusok esetén viszont futás alatti kötés (*runtime binding*) történik, azaz csak futás közben derül ki, hogy a hierarchia mely konkrét osztályának adott nevű metódusa fog lefutni. E technikának köszönhető, hogy mindig a hívó objektum osztályának metódusa hívódik meg (lásd 3.4. ábra). Figyelem, amikor egy virtuális vagy dinamikus metódust felülírunk egy utód osztályban, kötelező használnunk az *Override* direktívát (úgy, ahogy a TP-ben a *virtual* szót kellett mindig ismételni). Ha nem használjuk az *Override* direktívát, akkor a metódus statikusként fog működni.
De vajon mi a különbség a virtuális és dinamikus metódusok között? A különbség a futás alatti kötést megvalósító háttértechnikából ered. A virtuális metódus címe a *VMT*-ből (*Virtual Method Table*), míg a dinamikusé a *DMT*-ből (*Dynamic Method Table*) származik. Egy osztály *VMT*-jében megtalálhatók az addig a szintig definiált összes virtuális rutinok (a ténylegesen felülírtak és a csak örökölt metódusok) címei; egy adott metódus címe mindig ugyanazon a relatív sorszámon található: ha ez egy felülírt metódus, akkor a cím az aktuális osztály metódusára mutat, ha viszont ez egy örökölt és változatlanul hagyott metódus, akkor a címe egy ős osztálybeli metódusra fog mutatni. Megállapíthatjuk tehát, hogy az öröklési láncban a *VMT* egyre csak növekszik.
Ezzel szemben egy osztály *DMT*-jében csak a felülírt dinamikus metódusok címei szerepelnek. A DMT tehát mindig annyi bejegyzést tartalmaz, ahány ténylegesen felülírt metódus található az adott osztályban: minden bejegyzés megfelel egy felülírt dinamikus metódusnak, az ő címét tartalmazza. Egy csupán örökölt és változatlanul hagyott dinamikus metódus címe egy felsőbb DMT-ből fog kikerülni, pontosabban annak az ős osztálynak a DMT-jéből, ahol utoljára felül lett definiálva.
Tehát, ha futás közben egy virtuális rutin címét keressük, akkor azt az aktuális objektum osztályának *VMT*-jéből nézzük ki (ott biztosan meg is fogjuk találni, függetlenül attól, hogy ez a cím hova mutat). Ha viszont egy dinamikus metódus címét szeretnénk megtudni, akkor keresgélni kell az aktuális osztály *DMT*-jében, ha ott nincs, akkor az ős *DMT*-jében, és mind így fölfelé a hierarchiában mindaddig, amíg megtaláljuk. Biztosan meg fogjuk találni, hiszen egyébként már fordításkor hibát észleltünk volna, de ki tudja mikor, mennyi keresgélés után.

A virtuális metódusok használatánál a hívás gyorsabb, azonban nagyobb a memóriaigénye. A dinamikusoknál kevesebb memória fogy, de lassúbb a hívás.

[1] Statikus metódus: egyértelmű („beégetett") címet jelent a fordító számára.



Mikor használjunk dinamikus és mikor virtuális metódusokat?
Ha előreláthatólag egy metódust az utódosztályokban gyakran felül fognak írni, akkor azt érdemes dinamikusnak deklarálni. Ez esetben, ha egy utódosztályban meghívjuk a metódust, akkor a megoldás valahol „a közelben" lesz: vagy abban a konkrét utódosztályban, vagy az ősénél, de nem kell a sokadik őséig mennie. Ilyen például a *Click*, *MouseDown*, *DblClick*...
Ha egy metódust nagyon kis valószínűséggel fognak felülírni – de ugyanakkor meg akarjuk adni a felülírás lehetőségét –, akkor jobb, ha azt virtuálisnak deklaráljuk. Ilyen például a *GetDeviceContext*, mely már a *TControl* osztályban megjelenik, és minden utódja változatlan formában használja (pl. a képernyőre történő kiíráshoz).
Más megközelítésből azt is mondhatjuk, hogy egy metódust akkor ajánlatos dinamikusnak deklarálni, ha a metódust csak ott hívják meg, ahol felülírják. Gondoljunk csak a *Click* dinamikus metódusra: ha nem akarunk ráépíteni semmit, akkor nem írjuk felül, és nem is hívjuk meg. S mivel dinamikus, a *DMT*-ben nincs bejegyzése, nem foglalja fölöslegesen a helyet. Ha viszont szükségünk van rá, akkor felülírjuk, és természetesen meg is fogjuk hívni. Ekkor az új rutin címe az aktuális osztály *DMT*-jéből kerül ki, így a hívása gyors lesz.

Absztrakt metódus

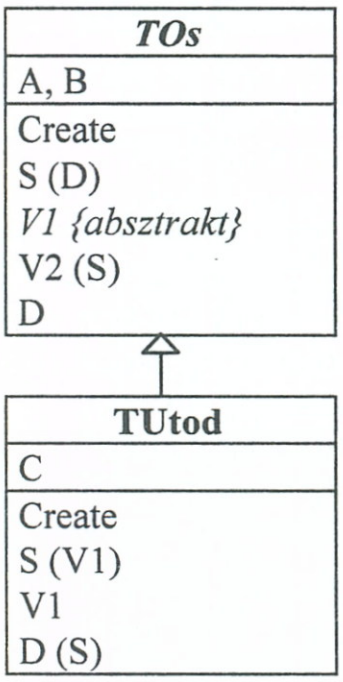
Az Object Pascal osztálymodellben lehetőség van „igazi" absztrakt metódusok deklarálására. Ezt az *abstract* fenntartott szó segítségével tesszük. Az absztrakt osztályban a rutin kifejtését nem kell megadnunk, később, az utód osztályokban kell felülírnunk, implementálnunk. Emlékeztetőül: **az absztrakt metódusok üres, virtuális metódusok, melyek csak örökítési célokat szolgálnak.** Tehát pont az a lényeg bennük, hogy nincs implementációjuk, ezt majd később, az utódosztályokban kapják meg. Természetesen egy absztrakt metódusokat tartalmazó osztálynak nem lehetnek példányai.
TP-ben a fordítót kénytelenek voltunk „becsapni" egy üres metóduskifejtéssel (*Begin End;*). OP-ben a fordító „ért a szóból": ha *abstract*-ként deklarálunk egy metódust, akkor annak kifejtését nem hiányolja.
Az absztrakt rutint vagy virtuálisnak, vagy dinamikusnak kell deklarálnunk, csak így van értelme az egésznek (a fordító is hibát jelez, ha ezt nem tesszük).

Nézzünk most mindezekre egy példát:

```
Type
   TOs = Class
      A,B: Word;
      Constructor Create(IA, IB:Word);
      Procedure S;
      Procedure V1; Virtual; Abstract;
      Procedure V2; Virtual;
```

```
      Procedure D; Dynamic;
   End;
   TUtod = Class(TOs)
      C: Char;
      Constructor Create(IA,IB:Word; IC:Char);
      Procedure S;
      Procedure V1; Override;
      Procedure D; Override;
   End;
...
Constructor TUtod.Create(IA,IB:Word; IC:Char);
   Begin
      Inherited Create (IA, IB);
      C := IC;
   End;...

Var
   Utod: TUtod;
Begin
   ...
   Utod.V2;
   ...
End.
```

Definiáltunk egy *TOs* nevű osztályt: van két adatmezője, konstruktora, egy statikus, két virtuális és egy dinamikus metódusa. Ebből származtattunk egy *TUtod* osztályt, melyben néhány metódust felülírtunk (*override!*). A következő ábra a feladat osztálydiagramját mutatja. Zárójelben feltüntettük az adott metódusból hívott másik metódus nevét.



3.4. ábra. Osztály-hierarchia

Ha egy utód példánynak meghívjuk a *V2* metódusát, akkor a következő rutinok kerülnek végrehajtásra:

```
TOs.V2, TOs.S, TUtod.D, TUtod.S, TUtod.V1
```

Az utód osztály *VMT*-jében a *TOs.V2* címe szerepel, hiszen ez a *TUtod*-ban nincs felülírva. Továbbá, meghívódik a *TOs.S*, mivel ez egy statikus rutin és már fordításkor be lett égetve a címe. A *D* nevű metódus címe a *TUtod DMT*-jében szerepel, tehát a *TUtod.D* hívódik meg. Utoljára az utódbeli *S* fut le, ő is statikus.

Működésileg tehát semmi különbség sincs a virtuális és dinamikus metódusok között.

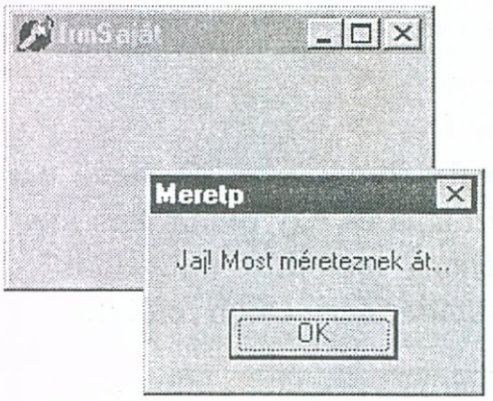
Egy másik érdekesség: ha valamilyen okból kifolyólag meghívásra került volna a *TOs.V1* absztrakt metódus, akkor egy futási hiba figyelmeztetett volna ennek helytelenségére.

Eseménykezelő metódus (*Message handler*)

Ne feledjük egy pillanatra sem, hogy windowsos környezetben vagyunk. Lehetőségünk van testre szabni különböző események kezelését. Ráállíthatunk egy metódust egy adott eseményre, azaz ahányszor az esemény bekövetkezik, a ráépített saját metódusunk fog lefutni. Minden Windows üzenet rendelkezik egy azonosító számmal; ennek vagy a hozzárendelt konstansnak ismeretében már csak a metódust kell megírnunk és ráirányítanunk a kívánt üzenetre.

```
{🖫 3_MERET\MERETP.DPR}
Type
   TfrmSajat = Class(TForm)
      Protected
         Procedure Atmeretezes ( Var Msg:TMessage); Message WM_SIZE;
   End;
...
Procedure TfrmSajat.Atmeretezes;
   Begin
      ShowMessage('Jaj! Most méreteznek át...');
      Inherited;
   End;
...
```

Az eredményt a 3.5. ábra szemlélteti.



3.5. ábra. Átméretezéskor a program „felkiált"

Minden eseménykezelő metódus rendelkezik egy változó paraméterrel: *Var Msg: TMessage*. Ez maga a bekövetkezett esemény üzenetté alakítva, azaz egy rekordszerkezet, melynek mezői tartalmazzák a konkrét eseményre vonatkozó adatokat (így ezeket fel is lehet használni).
Példánkban egy olyan rutint építettünk a *WM_SIZE* üzenetre, mely megjelenít egy üzenetablakot. Metódusunk az űrlap (*frmSajat*) minden egyes átméretezésénél meg fog hívódni. Ekkor megjeleníti az üzenetet (*Showmessage(...)*), majd az „*Inherited*" kulcsszó segítségével meghívja az ősosztálybeli átméretező kódot. De, hogy is van ez? TP-ben és OP-ben is az *Inherited* kulcsszó segítségével meghívhatunk egy ősosztálybeli metódust, de általában a kulcsszó után meg kell adnunk a metódus nevét és aktuális paramétereit (Pl. *Inherited Create(...)*). Az eseménykezelő metódusok esetén mindenképpen meg kell hívni az ős osztályban megírt és ugyanerre az üzenetre ráállított rutint, „hadd mondja el ő is a magáét", történjen meg az is, amit ott írtak meg. Igen ám, de vajon az ős osztályban hogyan nevezték el ezt a rutint? Ezt nem tudjuk; viszont kicsit alaposabban megvizsgálva a helyzetet rájövünk, hogy nem is érdekel, hiszen most a rutint nem a neve azonosítja, hanem az a tény, hogy a mi általunk „meglovagolt" üzenetre van ő is ráépítve. Így hát a fordító elfogadja, ha egyszerűen azt írjuk be, hogy *Inherited*, ő kikeresi számunkra a kívánt rutint (hiszen tudja az üzenet számát), és beépíti a hívását a saját metódusunkba.

3.3.4 Az adathozzáférés korlátozása

Az objektum adatmezőire ne hivatkozzunk közvetlenül.

Egy objektum adataira nem tanácsos kívülről közvetlenül hivatkozni. Az előre megírt metódusokat hívjuk, így egészen biztosan adataink nem fognak elromlani (legalábbis ha jól írjuk meg a metódusokat). Igen ám de a „nem tanácsos" és a „nem szabad" között óriási különbség van. Jó lenne, ha valamilyen úton-módon korlátozni tudnánk az adathozzáférést. Íme az Object Pascalban beállítható láthatósági szintek (3.6. ábra):

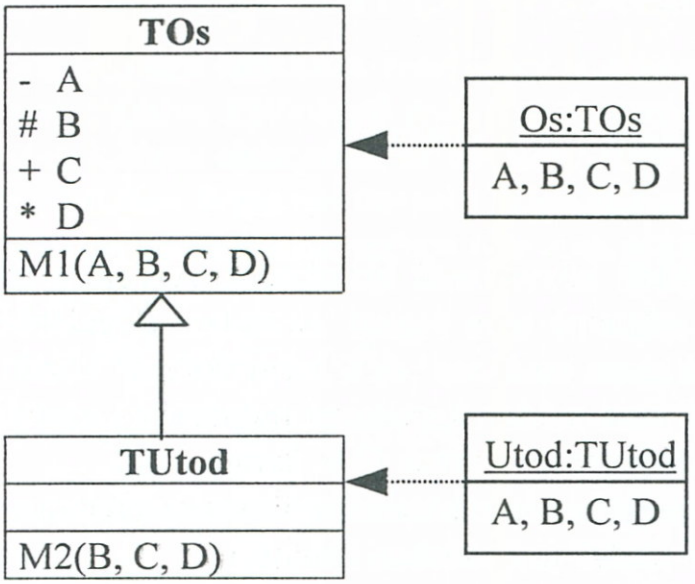
- **Private (privát)** adatok, jellemzők és metódusok: csak az illető osztály metódusaiból érhetők el. (Ilyen az *A* mező, mely csak az *M1* metódusban látható.)
- **Protected (védett)** adatok, jellemzők és metódusok: a kurrens osztály és az ebből származtatott osztályok metódusaiban láthatók. (Ilyen a *B* mező, mely elérhető az *M1* és az *M2* metódusban is.)
- **Public (nyilvános, publikus)** adatok, jellemzők és metódusok: kívülről, azaz objektumból is elérhetők. (Ez a *C* mező esete; elérhető az *M1* és *M2* metódusokból, de kívülről is: *Os.C*, *Utod.C*.)
- **Published (publikált)** metódusok és jellemzők: olyanok, mint a nyilvánosak, de már tervezés közben is állíthatók (az objektum-felügyelőben). Ilyen például a gombok felirata, betűtípusa... (Példánkban a *D* mező ilyen.)

  Csak a metódusokat és a jellemzőket deklarálhatjuk publikáltnak. A jellemzőkről a következő pontban lesz szó.

Mindezek az osztályon belül tetszőleges számban és sorrendben helyezkedhetnek el, egy csoportban előbb az adatok, majd a metódusok.

Vajon a tervezési időben űrlapra helyezett gombok, szerkesztődobozok... milyen láthatósági szinttel rendelkeznek? Mivel azt tapasztaljuk, hogy ezek már tervezési időben is láthatók, jellemzőik, eseményeik állíthatók, csakis publikáltak lehetnek.

Tekintsünk át egy példát (3.6. ábra)! Az UML jelölés szerint '–' jelzi a privát szintet, '#' a védettet és '+' a nyilvánost. Az UML-ben nincs jele a publikált adatoknak, így a '*' a szerző saját jelöléseként szerepel.
Tapasztaljuk, hogy kívülről csak a nyilvános vagy publikált adatokra lehet hivatkozni (a többinél „robban a pokolgép"), míg öröklési úton a védett adatok is elérhetők. A publikált adatok már tervezési időben is olvashatók, írhatók.




```
{Kódrészlet}
Var Os:TOs;
    Utod:TUtod;

...
Os.A := 10;
Os.B := 20;
Os.C := 30;
Os.D := 40;
Utod.A := 10;
Utod.B := 20;
Utod.C := 30;
Utod.D := 40;
```

3.6. ábra. Adathozzáférés korlátozása példányból és öröklési úton

A láthatóság korlátozásának hatásköre

Az Object Pascalban (akárcsak a TP-ben) a védelem egységszintű. Ha két osztály ugyanabban az egységben van kifejtve, akkor egymás adataira minden nehézség nélkül tudnak hivatkozni, figyelmen kívül hagyva a beállított védelmi szintet. Ez helyettesíti a C++ nyelvbeli *Friend* fogalmat, azaz az osztályok ebben az értelemben egymás barátai, „bizalmasai".

A láthatósági szint megváltoztatása

Lehet-e egyáltalán változtatni egy osztály mezőinek láthatóságát? És ha lehet, akkor milyen irányba: bővíteni vagy szűkíteni?
Öröklési úton az utód osztályban **az örökölt jellemzők láthatóságát bővíthetjük**. Ez azt jelenti, hogy egy védett vagy nyilvános jellemzőt újradeklarálhatunk például publikáltként. Ennek hatására a jellemző az utód osztály példányaiban már tervezési időben is elérhetővé válik. Visszafokozni azonban nem tudjuk a láthatóságot, ezt csak bővíteni lehet. Bővebben lásd a 15. fejezetben.

3.3.5 Jellemzők (*Properties*)

Az Object Pascal osztálymodell talán leglényegesebb újdonsága a jellemzők bevezetése. Tekintsük át ezeket egy példán keresztül:

```
Type TMyComponent = Class(TComponent)
    Private
      FJegy: Byte; {csak 1 és 5 közötti lehet}
    Public
```

```
      Function GetJegy: Byte;
      Procedure SetJegy(AValue: Byte);
   End;

Function TMyComponent.GetJegy: Byte;
begin
  Result := FJegy;
end;

Procedure TMyComponent.SetJegy(AValue: Byte);
begin
  If (AValue In [1..5]) And (AValue <> FJegy) Then
    FJegy:= AValue;
end;

...
```

Tegyük fel, hogy van egy osztályunk (egy komponensünk), melyben – többek között – tároljuk egy hallgató jegyét is valamilyen tantárgyból. Hagyományos objektumorientált eszközökkel felvennénk számára osztályunkban egy mezőt, az *FJegy*-t. Egy jegy elfogadható értékei 1 és 5 közöttiek. Ahhoz, hogy kivédjük a helytelen értékek beállítását, a mezőt privátnak kell deklarálnunk. Írnunk kell ugyanakkor két nyilvános metódust: a *SetJegy* segítségével beállíthatjuk a jegyet, a *GetJegy* segítségével pedig lekérdezhetjük az előzőleg beállított értéket. Természetesen a *SetJegy*-ben csak akkor állítjuk át a privát mezőt, ha az új érték a megengedett határokon belül esik.
Ha később létrehozunk egy objektumot a *TMyComponent* osztály mintájára, akkor ennek *FJegy* mezőjét a következőképpen állíthatjuk be:

```
Var MyComponent:TMyComponent;
...
MyComponent.SetJegy(3);   {FJegy ← 3}
MyComponent.SetJegy(7); {semmi sem történik}
...
```

Ennek a hagyományos megoldásnak több hátránya is van: első sorban a mező értékeinek állítása és lekérdezése kényelmetlen, hiszen speciális metódusokat kell hívogatnunk. Másodszor, ezt az adatot nem áll módunkban már tervezési időben állítani. Ahhoz, hogy a jegyet (*FJegy* mezőt) már tervezéskor állíthassuk, ennek publikáltnak kellene lennie. Ha pedig publikált, akkor mindenki hozzáfér, tehát elvileg nincs védelem. Gyakorlatilag azonban Object Pascalban egy sima adatot nem is deklarálhatunk publikáltnak (ezt az előző pontban láttuk). Most már azt is értjük, mi ennek az oka.

Mindezen problémákra megoldást jelent a jellemző bevezetése. Deklaráljunk egy kívülről is elérhető (nyilvános vagy publikált) jellemzőt:

```
Type TMyComponent = Class(TComponent)
    Private
      FJegy:Byte;
```

```
    Function GetJegy: Byte;
    Procedure SetJegy(AValue: Byte);
  Published
    Property Jegy:Integer Read GetJegy Write SetJegy;
 End;

Function TMyComponent.GetJegy: Byte;
begin
  Result := FJegy;
end;

Procedure TMyComponent.SetJegy(AValue: Byte);
begin
  If (AValue In [1..5]) And (AValue<>FJegy) Then
    FJegy:= AValue;
end;

...
```

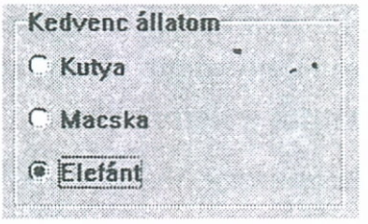
A jellemzőt a *property* szóval vezetjük be. Meg kell adnunk a jellemző nevét és típusát, majd a *Read* és *Write* fenntartott szavak után be kell írnunk annak a metódusnak a nevét, mely automatikusan le fog futni, valahányszor a jellemző értékét olvassák, vagy állítják. A *Jegy* jellemző lekérdezésékor az értéket a *GetJegy* függvény szolgáltatja, átállításakor pedig lefut a *SetJegy*. Az író/olvasó metódusok már privátok is lehetnek, hiszen ezeket kívülről nem hívjuk közvetlenül. Egy konkrét objektum jegyét most a következőképpen állítjuk be:

```
Var MyComponent:TMyComponent;
...
MyComponent.Jegy:=3;   {FJegy ← 3}
MyComponent.Jegy:=7; {semmi sem történik}
...
```

A jellemző értékét most már tervezési időben is látjuk, állíthatjuk, és természetesen a védelem ekkor is él. A *SetJegy* metódus automatikusan lefut, akár tervezési, akár futási időben állítjuk a *Jegy* jellemző értékét.

A felhasználó (az osztály felhasználója) szempontjából a jellemző olyan, mintha közvetlenül az adatot állítaná, a háttérben automatikusan lefutó olvasó/író (***access/assign, read/write***) metódusok számára láthatatlanok és kikerülhetetlenek. **A jellemző tulajdonképpen csak egy maszk, melyen keresztül a legtöbbször egy privát adatot állítunk és olvasunk.**

Vegyünk egy másik példát: egy választógomb-csoportban a pötty mindig csak egyik választási lehetőségben lehet. A pötty pozícióját az *ItemIndex* jellemző tárolja; értéke 0 ha a pötty a kutyánál van, 1 ha a macskánál, és 2 ha az elefántnál. Amikor ezt a jellemzőjét módosítjuk, mondjuk 2-ről 0-ra, akkor nem csak a tényleges adatnak kell megváltoznia, hanem a pöttynek is át kell kerülnie az egyik helyről a másikra. Ezt a frissítést, újrarajzolást a jellemző író metódusában kell megvalósítani.



3.7. ábra. Választógomb-csoport

Röviden tehát azt mondhatjuk a jellemzőkről, hogy:

- Általában minden jellemző mögött egy privát adatmező áll. A felhasználó ennek értékét állítja és olvassa az író/olvasó rutinok segítségével.
- A jellemző legtöbbször vagy nyilvános, vagy publikált. Privát jellemzőket nem túl értelmes dolog létrehozni, hiszen akkor pont a felhasználó – az, aki számára készültek – nem fogja ezeket elérni. Védett jellemzőt pedig csak akkor hozzunk létre, ha öröklési úton láthatóságát majd bővíteni szándékozzuk.
- Tervezés és futás közben csak a jellemző értéke állítható, olvasható, maga a tényleges adat rejtve marad.
- Az olvasás/írás az osztály metódusaival történik, melyek általában privátok. Ez azt jelenti, hogy a felhasználónak abszolút nincs beleszólása ezen metódusok működésébe. Ez kimondottan szerencsés tény, hiszen egy választógomb-csoport esetén (3.7. ábra) mindig ugyanúgy kell törlődnie a pöttynek az egyik helyről, és ugyanúgy kell megjelennie az új választásnál. És ha lehet, akkor a felhasználó ne is lássa, hogy ez hogyan történik, fölösleges őt ezzel terhelni. Nem csak az adatokat lehet tehát jellemzőkkel elrejteni, hanem a megvalósítási, működési részleteket is.
- Írásnál (a *Write* metódusban) beépíthetünk értékellenőrzéseket (mint példánkban), frissítéseket, újrarajzolásokat (a választógomb csoportnál a pötty törlése és újrarajzolása), számolásokat (például, ha egy jelszót titkosítva tárolunk, akkor a *SetJelszo* végezné a titkosítást, és a jelszó így kerülne tárolásra a privát mezőben; olvasásnál a *GetJelszo* metódusnak kellene a dekódolást elvégeznie. Mindkét metódus láthatatlan lenne a felhasználó számára, nem is tudna az adat titkosításáról).
- Olvasásnál beépíthetünk számolásokat (például ha az adat titkosítva kerül tárolásra).
- A publikált jellemzők értékeit már tervezéskor az objektum-felügyelőben is látjuk, állíthatjuk (ilyen a gomb mérete, pozíciója, felirata, szerkesztődoboz szövege...). Állításuknál és olvasásuknál automatikusan lefutnak a megfelelő író/olvasó metódusok, és ezeknek köszönhetően az adatok elronthatatlanok.

Próbálja meg beállítani egy gomb magasságát -15-re vagy „tizenöt”-re. Mi történik?

Ha valamelyik irányba (íráskor vagy olvasáskor) nem akarunk semmilyen különlegesebb tevékenységet elvégezni (példánkban a *GetJegy*), akkor a következőképpen is deklarálhatjuk jellemzőnket:

```
Property FJegy: Byte Read FJegy Write SetJegy;
```

Ez esetben nincs különösebb olvasási tevékenység, tehát a felhasználó közvetlenül a privát adat értékét látja. Ugyanezt megtehetjük az írásnál is, ekkor az író metódus neve helyett a privát adatot tüntetjük fel. Annak azonban, hogy se író, se olvasó metódust ne adjunk, nincs különösebb értelme, hiszen akkor inkább deklaráltuk volna publikáltnak magát az eddigi privát adatot.
Mivel az olvasás legtöbbször „ártatlan" művelet, az írás viszont annál veszélyesebb, általában a jellemzők olvasása közvetlenül az adatmezőből történik (nem adunk meg függvényt), az írást pedig egy igen jól kigondolt és megírt eljárással valósítjuk meg.

Ha egyáltalán nem deklarálunk olvasó vagy író részt (akár magát az adatot, akár egy metódust), akkor az adat csak írható, illetve csak olvasható lesz. Nyilvánvaló, hogy általában nem kívánunk csak írható (de nem olvasható) adattal dolgozni, tehát sose fogunk egy jellemzőt így deklarálni:

```
Property Count: Integer Write SetCount;
```

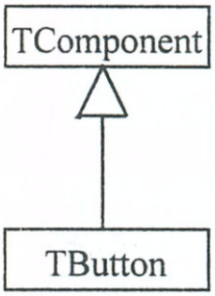
3.3.6 Osztályoperátorok

Az Object Pascalban két olyan operátort vezettek be, melyeknek operandusai objektumok és osztálynevek: ***Is*** és ***As***. Az operátorok által elvégzett műveleteket a TP-ben is meg lehetett valósítani, de az Object Pascalban mindez hatékonyabban valósítható meg.
Az ***Is*** operátor egy objektum típusának (fajtájának) a megállapítására való. Formája:

```
If Obj Is TButton Then
  ShowMessage('Ez egy gomb');
```

Erre használhattuk a TP-ben a *TypeOf* függvényt. A kettő közötti különbséget a 3.8. ábra szemlélteti.

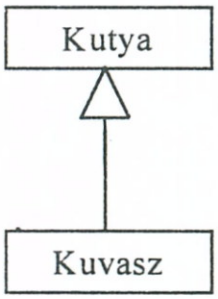
Az objektumot *TComponent*-nek deklaráltuk, inicializálásnál viszont már *TButton*-ként hoztuk létre. Az **Is** mindkét típust „felismeri", ezek kompatíbilisek. A TP-beli *TypeOf* függvény viszont már csak *TButton* típusúnak látja az objektumot, elfelejtve a deklarációját.




```
Var Obj: TComponent;
...
Obj := TButton.Create(...);
```

Típusösszehasonlítás	**Is** (fajtája)	TypeOf (típusa)
Obj, TButton	True	True
Obj, TComponent	True	False

3.8. ábra. *Is* és *TypeOf* összehasonlítása



Bodri egy kuvasz. Tágabb értelemben ő egy kutya.

Típusösszehasonlítás	**Is** (fajtája)	TypeOf (típusa)
Bodri, kuvasz	True	True
Bodri, kutya	True	False

3.9. ábra. Analógia a valós életből

Az *As* operátor típuskonverzióra használható. Formája:

```
With Obj As TButton Do
   Begin
      Caption := 'Ok gomb';
      OnClick := OkClick;
   End;
...
{vagy}
(Obj As TButton).Caption := 'Ok gomb';
```

A két operátort általában együtt használjuk. Csak akkor kényszerítünk egy adott típust egy objektum-mutatóra, ha előtte meggyőződtünk arról, hogy ő most konkrétan tényleg olyan típusú objektumra mutat. Ha véletlenül más, nem kompatíbilis típust kényszerítenénk egy objektumra, akkor az *EInvalidTypeCast* kivétel képződik. A kivételekkel a 3.5. pontban ismerkedünk meg.

```
Var Obj:TComponent;
...
   If Obj Is TButton Then
      (Obj As TButton).Caption := 'Ok gomb';
...
```

Futtassa le a 3_OSZTALYOP\POPERATOR.DPR példaprogramot! Az űrlapon látható gomb és szerkesztődoboz kattintása ugyanabba az eseménybe torkollik. Ebben lekérdezzük az esemény forrását (gomb vagy szerkesztődoboz, és annak állítjuk át a feliratát. Ehhez mindkét ismertetett osztályoperátort alkalmazzuk.

3.4 Összetett típusú függvényértékek

Object Pascalban a függvények bármilyen típusú adatot visszaadhatnak az *Object* (ez még a TP-beli osztályok típusa) és az állománytípusokon kívül[1].
A függvénynek értéket adhatunk a függvényazonosítóval, vagy az újonnan bevezetett **Result** segítségével. A *Result* a függvényben lokális változóként viselkedik; a fordító automatikusan beépíti a deklarációját, majd a függvény legvégén a *függvényazonosító:=Result* értékadást.
Mit nyerünk a *Result* használatával? Először is kódunk áttekinthetőbbé válik, másodszor pedig a *Result* használható értékadás jobb oldalán is, anélkül, hogy rekurzív híváshoz vezetne.
Például:

```
type TComplex = record
        Re,Im:Real;
     end;

function Add(C1,C2:TComplex):TComplex;
begin
   Result.Re:=C1.Re+C2.Re;
   Result.Im:=C1.Im+C2.Im;
end;

function Hatvany(var X:integer; N:Byte):LongInt;
var i:Byte;
begin
   Result:=1;
   For i:=1 to N Do
      Result:=Result*X;
end;
```

3.5 Kivételek kezelése *(Exception handling)*

A magasszintű nyelvek új változataiban két irányelv figyelhető meg:

- „**No pointers**", azaz a mutatók automatikusan épüljenek bele programjainkba, anélkül, hogy nekünk „bíbelődnünk" kellene velük. Ez Delphiben is így van: amikor deklarálunk pl. egy *TButton* típusú gombot (*Gomb:TButton*), akkor tulajdonképpen egy gomb típusú mutatót vezetünk be; ugyanakkor a *Gomb* feliratát simán, kalapozás nélkül elérjük, állíthatjuk: *Gomb.Caption* (bővebben tárgyaltuk a 3.3. pontban).
- A másik irányelv a normál programlogika elkülönítése a hibalogikától. Ez azt jelenti, hogy rutinunkat az általános esetre írjuk meg, figyelmen kívül hagyva a határeseteket, majd a rutin végén felsoroljuk a különböző hibalehetőségeket, és tennivalókat. Így

[1] Turbo Pascalban csak sorszámozott, valós, karakterlánc vagy mutató típussal térhet vissza egy függvény.

sokkal könnyebb a programozás, hiszen nem kell minden utasításnál vizsgálgatni a lehetséges hibákat. Elég erre a rutin végén figyelni. Mindemellett a kód is sokkal áttekinthetőbb.

Object Pascalban a hibakezelés az ún. **kivételek (*Exceptions*)** segítségével valósul meg. **A kivétel olyan hibás állapot vagy esemény, mely megszakítja az alkalmazás futását.** Delphiben minden futás idejű hiba kivétellé alakul át. Ezt programunkban észrevehetjük egy speciális utasítás segítségével, és feldolgozhatjuk a megfelelő módon; ha ezt nem tesszük, akkor az alapértelmezett kivételkezelő fog lefutni, mely a futási hiba angol szövegét jeleníti meg egy kis ablakban.

A Delphi futtató rendszerben e kivételek megjelenésekor lehetőség van leállítani a rendszert (ez olyan mintha a hibát okozó utasításnál egy töréspontot helyeznénk el). Beállítási helye: *Tools/ Environment Options/ Break on Exception* jelölőnégyzet. Kipipálása az alkalmazások fejlesztése közben javasolt, ekkor segíti a programozót a hibák megtalálásában.
Ha viszont alkalmazásunkat nem a Delphi környezetből futtatjuk, akkor e beállításnak semmi hatása, a kivétel képződésekor programunk nem fog leállni a hibás soron.

A kivétel tulajdonképpen egy hibaobjektum, mely a futási hiba következtében keletkezik, és feldolgozása után automatikusan megszűnik. Osztálya egy a sok kivételosztály közül (3.10. ábra).

```
TObject
└─Exception
    ├─ EOutOfMemory          {elfogyott a memória}
    ├─ EInOutError           {I/O hiba}
    ├─ EIntError             {egész számokat érintő hibák}
    │   ├─ EDivByZero        {nullával osztás}
    │   ├─ ERangeError       {a jó értékhatáron kívüli érték}
    │   └─ EIntOverflow      {túlcsordulás}
    ├─ EMathError            {valós számokat érintő hibák}
    │   ├─ EZeroDivide       {nullával osztás}
    │   ├─ EOverflow         {túlcsordulás}
    │   ...
    ├─ EConvertError         {konverziós hiba}
    ├─ EAccessViolation      {általános védelmi hiba}
    ├─ EStackOverflow        {verem túlcsordulás}
    ├─ EPrinter              {nyomtató hiba}
```

3.10. ábra. Kivételosztályok a Delphi osztályhierarchiában

Minden kivételosztályban van egy **Message:String** mező. Ez tartalmazza a hiba angol szövegét.

A futási hiba keletkezésekor két dologra kell gondolnunk:

- Le kell kezelnünk a hibát a megfelelő módon (3.5.1. pont).
- Fel kell szabadítanunk a hiba előállta előtt lefoglalt erőforrásokat: memóriát felszabadítani, nyitott állományokat bezárni... (3.5.2. pont).

3.5.1 Védelem a futás-idejű hibák ellen (*Try...Except*)

A kivételek figyelése és kezelése a következő utasítás segítségével történik:

```
Try                    {próbáld végrehajtani a védett blokk utasításait}
  utasítás blokk                          {védett blokk, protected block}
Except                        {ha pedig valamilyen hiba történt, akkor...}
  On KivételOsztály Do       {ha ez a hiba, akkor hajtsd végre ezt a }
    Kivételkezelő blokk                          {kivételkezelő blokkot}
  ...                                                              {...}
[Else                      {minden egyéb hiba esetén, hajtsd végre ezt a}
  kivételkezelő blokk]                           {kivételkezelő blokkot}
End;
```

Tekintsünk át egy átlagszámolást hagyományos és új programozási módszerekkel! Az összeget az *eOsszeg*, a darabszámot az *eDb* szerkesztődobozokban kérjük be, míg az átlagot az *eAtlag*-ban jelenítjük meg.

```
{🖫 3_KIVETELEK\PKIVETEL.DPR }
procedure TfrmSzamolas.SzamolHagyomanyosanClick(Sender: TObject);
var  Osszeg:Real;
     Db, Kod:Integer;
begin
  Val(eOsszeg.Text, Osszeg, Kod)[1];
  If Kod <> 0 Then
    ShowMessage('Az összegnek valósnak kell lennie.')
  else
    begin
      Val(eDb.Text, Db, Kod);
      If Kod <> 0 Then
        ShowMessage('A DB-nek egésznek kell lennie.')
      Else
        begin
          If Db=0 Then
            ShowMessage('A Db nem lehet nulla.')
          Else
```

```
                eAtlag.Text:= FloatToStr(Osszeg/Db);
            end;
        end;
end;

procedure TfrmSzamolasKivetelKezelessel.SzamolClick(Sender: TObject);
var  Osszeg:Real;
     Db:Integer;
begin
   Try
      Osszeg:= StrToFloat(eOsszeg.Text);
      Db:= StrToInt(eDb.Text);
      eAtlag.Text:= FloatToStr(Osszeg/Db);
   Except
      On EConvertError Do
         Showmessage('Az összegnél valós, a Db-nél pedig egész '+
                     értéket írjon be.');
      On EZeroDivide do
         ShowMessage('A Db-nek nullától különbözőnek kell lennie.');
   End;
end;
```

A kivételkezeléses megoldásban használjuk a *FloatToStr* és *StrToFloat* konverziós függvényeket. Ezek elvégzik a konverziót, ha ez lehetséges. Ellenkező esetben *EConvertError* típusú kivételobjektumot generálnak. Ezt mi a *Try...Except* utasítás segítségével észrevesszük és feldolgozzuk. Mennyivel rövidebb és áttekinthetőbb a második megoldás! Igaz, ebben nem különböztetjük meg az *Osszeg* és a *Db* típushibáját. Ahhoz, hogy a két esetet megkülönböztetve kezeljük, a két konverziót két külön *Try...Except* utasításba kell helyeznünk. Valahogy így:

```
procedure TfrmSzamolas. TfrmSzamolasKivetelKezelessel2(Sender:
                                                      TObject);
var  Osszeg:Real;
     Db:Integer;
begin
   Try
      Osszeg:= StrToFloat(eOsszeg.Text);
      Try
         Db:= StrToInt(eDb.Text);
         eAtlag.Text:= FloatToStr(Osszeg/Db);
      Except
         On EConvertError Do
            Showmessage('A Db-nek egésznek kell lennie.');
      End;
   Except
```

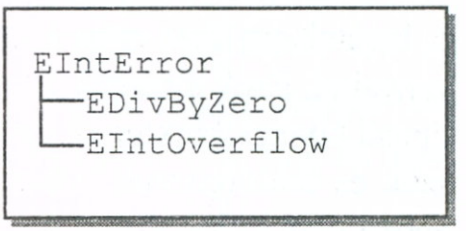
[1] TP-ben még a *Val* eljárás segítségével alakítottunk át egy karakterláncot számmá. OP-ben, mint tudjuk, erre van kényelmesebb megoldás is: *StrToInt*. Ezt alkalmazzuk a második megoldásban.

```
      On EConvertError Do
         Showmessage('Az összegnek valósnak kell lennie.');
      On EZeroDivide do
         ShowMessage('A Db-nek nullától különbözőnek kell lennie.');
   End;
end;
```

Azt vesszük észre, hogy a kivételkezelők egymásba is ágyazhatók.
Elvileg mindegy, hogy a nullával való osztás ellenőrzését melyik kivételkezelőbe tesszük. Ha a belsőben képződik, és ott nem dolgozzuk fel, akkor továbbterjed a külsőbe. Ha pedig ott sem dolgoznánk fel, akkor még mindig van, aki feldolgozza: lefut a rendszerbe beépített alapértelmezett kivételkezelő.

A *Try...Except* -ben a kivételosztályok felsorolásának sorrendje nem minden esetben tetszőleges. Akárcsak a *Case* utasításnál, a rendszer itt is a beírás sorrendjében ellenőrzi a kivételosztályokat; az első találathoz írt utasításokat végrehajtja, és nem keres tovább. Akkor van találat, amikor a vizsgált kivételosztály típus szerint kompatíbilis az aktuális hiba kivételével. Emiatt, előbb mindig a származtatott kivételosztályok kezelőit soroljuk fel.

Figyelembe véve a 3.11. ábrát tapasztaljuk, hogy a következő két programrészlet nem azonos hatású:



3.11. ábra

Ez a helyes sorrend	**Ez a rossz sorrend** (a második és a harmadik ág sose hajtódik végre)

`Try`
`  ...`
`Except`
`  On EDivByZero Do`
`  ShowMessage('Osztás 0-val');`
`  On EIntOverflow Do`
`  ShowMessage('Túlcsordulás');`
`  On EIntError Do`
`  ShowMessage('Egyéb egész'+`
`  ' aritmetikai hiba');`
`End;`
	`Try`
	`  ...`
	`Except`
	`  On EIntError Do`
	`  ShowMessage('Egyéb egész'+`
	`  ' aritmetikai hiba');`
	`  On EDivByZero Do`
	`  ShowMessage('Osztás 0-val');`
	`  On EIntOverflow Do`
	`  ShowMessage('Túlcsordulás');`
	`End;`

Ha a hiba feldolgozása mellett még a keletkezett hiba szövegét is meg szeretnénk jeleníteni, akkor azt a kivételobjektum *Message* jellemzőjéből olvashatjuk ki. Ebben az esetben szükség van az objektumra, nevezzük tehát el:

```
Try
   ...
Except
   On E:EConvertError Do
      begin
         {hibakezelés}
         ShowMessage(E.Message);
      end
End;
```

Az EAbort kivétel

Az *EAbort* egy speciális „halk" kivétel (*silent exception*). Akárcsak a többi kivétel ez is megszakítja a program futását, és a programblokkból való kilépéshez vezet, azonban az *EAbort* – a többi kivétellel ellentétben – le nem kezelése esetén sem jelenít meg hibaüzenetet. Ezért nevezik „halk" kivételnek. Az *EAbort* kivételt az *Abort* eljárás segítségével idézhetjük elő. Ezt akkor szoktuk alkalmazni, ha ki akarunk lépni az adott eljárásból, vagy meg akarjuk szakítani az eseményláncolatot (lásd később).

3.5.2 Erőforrások biztonságos használata (Try...Finally)

Ha például megnyitunk egy állományt, és feldolgozása közben hiba keletkezik, akkor a hiba lekezelésekor tanácsos az állományt bezárni. Ugyanez a helyzet a lefoglalt memóriaterületekkel is. Általában: ha egy erőforrást lefoglalunk, akkor minden esetben gondoljunk ennek felszabadítására is. Object Pascalban ezt a **Try...Finally** utasítás segítségével oldhatjuk meg.

```
erőforrások lefoglalása
Try
   erőforrások használata
Finally
   erőforrások felszabadítása
End;
```

Ha nem lép fel hiba, akkor minden lefut: erőforrások lefoglalása, használata, majd felszabadítása. Ha viszont az erőforrások használata közben hiba keletkezik, akkor a vezérlés rögtön átkerül a *Finally* részhez, az ide írt utasítások végrehajtódnak; a hibakezelés csak ezután kerül sorra, rendszerint egy külső *Try...Except* utasításban.

A *Finally* részbe írt utasítások minden esetben lefutnak a kivétel keletkezésétől függetlenül.

Tekintsük át az állományok kezelését hagyományos és új módszerekkel:
(🖫 3_KIVETELEK\PKIVETEL.DPR)

Állományműveletek hagyományos programozási módszerekkel

```
procedure TfrmSzamolas.
     AllomanyHagyomanyosanClick
     (Sender: TObject);

var  F: System.Text;
     S: String;

begin
   AssignFile(F, 'Proba.txt');
   {$I-}
   Reset(F);
   {$I+}
   If IOResult <> 0 Then
      ShowMessage('Az áll. nem
                   létezik')
   Else
      begin
         ...
         CloseFile(F);
      end;
end;
```

Állományműveletek kivételkezeléssel

```
procedure TfrmSzamolas.
   AllomanyKivetelekkelClick
   (Sender: TObject);

var  F: System.Text;
     S: String;

begin
   AssignFile(F,'Proba.txt');
   Try
      Reset(F);
      Try
         ...
      Finally
         CloseFile(f);
      End;
   Except
      On E:EInOutError Do
         ShowMessage(E.Message);
   End;
end;
```

A szöveges állomány típusazonosítója (*Text*) a *System* egységben van definiálva. Mindig minősítenünk kell az egység nevével, egyébként a fordító az *frmSzamolas:TfrmSzamolas* űrlap *Text* jellemzőjére „gondol".

Figyelem! A következő három kivétel csak akkor képződik, ha a fordítót – akár fordítási direktívával, akár környezeti beállítással – utasítjuk arra, hogy a megfelelő ellenőrző kódot építse be: *EInOutError* ⇔ *{$I+}*, *EIntOverflow* ⇔ *{$Q+}*, *ERangeError* ⇔ *{$R+}*.

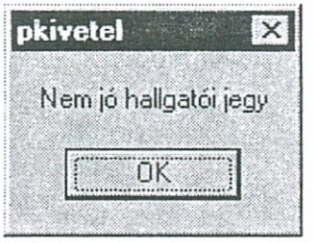
3.5.3 Saját kivételek létrehozása

Object Pascalban saját kivételeket is bevezethetünk. Lépések:

1. Hozzuk létre az új kivételosztályt. Őse az *Exception* legyen, csak így tudjuk majd kivételünket a *Try...Except* és *Try...Finally* utasításokkal kezelni.
2. A hiba keletkezésekor hozzuk létre egy példányát (***Raise*** utasítás).

Tekintsünk át egy példát! Egy hallgatói jegyet kérünk be az *eJegy* szerkesztődobozban. Ha minden OK, akkor kezdődhet a feldolgozás.

```
{ 3_KIVETELEK\PKIVETEL.DPR }
Type  ENemJoJegy = Class(Exception);
...
procedure TfrmSzamolas.SajatKivetelClick(Sender: TObject);
var Jegy:Byte;
begin
   Try
      Jegy:= StrToInt(eJegy.Text);
      If (Jegy<1) Or (Jegy>5) Then
         Raise ENemJoJegy.Create('Nem jó hallgatói jegy!');
      ...{jöhet a feldolgozás}
   Except
      On EConvertError Do
         Showmessage('A jegy csak egész szám'+
                     ' lehet');
      On E:ENemJoJegy Do
         ShowMessage(E.Message);
   End;
end;
```



3.12. ábra. Saját kivételünk

3.5.4 Kivételek ismételt előidézése

Bizonyos esetekben előfordul, hogy ugyanazt a kivételt többszörösen is fel szeretnénk dolgozni. Például, ha egy adott kivétel több helyen is előfordulhat programunkban, és kezelését egységesen szeretnénk megvalósítani, akkor a kivételek keletkezésekor helyben állítsunk be egy globális karakterláncot a hiba okára utaló szöveggel; a globális kivételkezelőben ezt a szöveget fogjuk megjeleníteni. Van azonban egy probléma: a kivételobjektum rögtön lekezelése után megszűnik. Miután beállítjuk a hibaszöveget, a belső kivételkezelőben, a hibaobjektum megszűnik. Ha azt szeretnénk, hogy egy külső *Try...Except* blokkban is kezelődjön le, akkor újra kell élesztenünk a ***Raise*** utasítással.

```
{ 3_KIVETELEK\PKIVETEL.DPR }
procedure TfrmSzamolas.KivetelIsmeteltLekezeleseClick(Sender:TObject);
Var  Mes:String;
begin
   Try
      {A belső kivételkezelőkben beállítjuk a Mes változóba a
      hibaüzenetet, és újraélesztjük a kivételt}
      Try
         ...
      Except
         On EZeroDivide Do
            Begin
               Mes := 'Nullával való osztás még a legelején';
               Raise;
            End;
      End;
```

```
      Try
      ...
      Except
         On EZeroDivide Do
            Begin
               Mes := 'Nullával való osztás a közepén';
               Raise;
            End;
      End;

      Try
         ...
      Except
         On EZeroDivide Do
            Begin
               Mes := 'Nullával való osztás a legvégén';
               Raise;
            End;
      End;
   Except
      {Az újraélesztett kivétel itt kerül sorra (a külső
      kivételkezelőben). Itt egységesen kezeljük le a belső hibákat.}
      On EZeroDivide Do
         ShowMessage(Mes);
   End;
end;
```

3.6 Object Pascal karakterláncok

A Delphi különböző verzióiban különböző típusú karakterláncokkal dolgozhatunk.

Delphi 1-es karakerláncok:

- **String:**

  Ez a hagyományos Pascal karakterláncnak felel meg: maximum 255 karaktert tartalmazhat, a nulladik bájtjában a rendszer a hosszát tárolja. Használhatók a megszokott karakterlánc-kezelő operátorok és függvények: értékadás (:=); összehasonlítások (<, >, <=, >=, =); konkatenálás (+); indexelés (S[5]); hosszlekérdezés (*Length(S)*)...

- **PChar** (=^Char) és **Array[0..Max] of Char**:

  A C típusú karakterlánc OP-beli megfelelői: max. 64 KB-osak lehetnek, a rendszer végjelként az utolsó karakter után egy 0 kódú karaktert helyez el. Használatuk az API rutinok hívásakor válik szükségessé. Kezelésük speciális függvények segítségével történik: helyfoglalás (*StrAlloc*, csak a *PChar*-nál van erre szükség); felszabadítás (*StrDispose,* az *StrAlloc* párja), értékadás (*StrCopy*); összehasonlítás (*StrComp*); konkatenálás (*StrCat*); hosszlekérdezés (*StrLen*)... (bővebben lásd a súgóban).

A Delphi 32 bites verzióiban bevezetett karakterláncok

A fentieken kívül a Delphi 2-ben a ***ShortString*** és ***AnsiString*** karakterlánctípusokat is használhatjuk. A Delphi 3-ban ezekhez hozzáadódik a ***WideString*** is. Vegyük mindezeket sorba:

- **ShortString**: max. 255 karaktert képes tárolni, azonos a hagyományos *String* típussal.
- **AnsiString**: 255 karakternél hosszabb karakterláncok tárolására alkalmas, maximális hosszának csak a rendelkezésre álló memória szab határt, elvileg max. 2GB.
- **WideString**: akárcsak az *AnsiString*, de karaktereit két bájton tárolja (Unicode karakterek).

Mindegyik új típusnál használhatók a megszokott karakterlánc-kezelő operátorok és függvények, lásd Delphi 1-ben a *String* típust. Egy korlátozás azonban létezik: az *AnsiString* és *WideString* típusú karakterláncok – a korlátlan hosszuk miatt – nem menthetők ki típusos állományokba. Ha tehát állományokkal dolgozunk, akkor karakterláncaink számára használjuk a *ShortString* típust.

A Delphi 32 bites verzióiban is használhatók a *String* és *PChar* típusok. A *PChar*-t itt is leginkább az API függvények hívásakor használjuk.

A *String*-ről fontos tudnunk azt, hogy nem mindig viselkedik hagyományos módon. Értelmezése a *{$H}* fordítási direktívától függ:

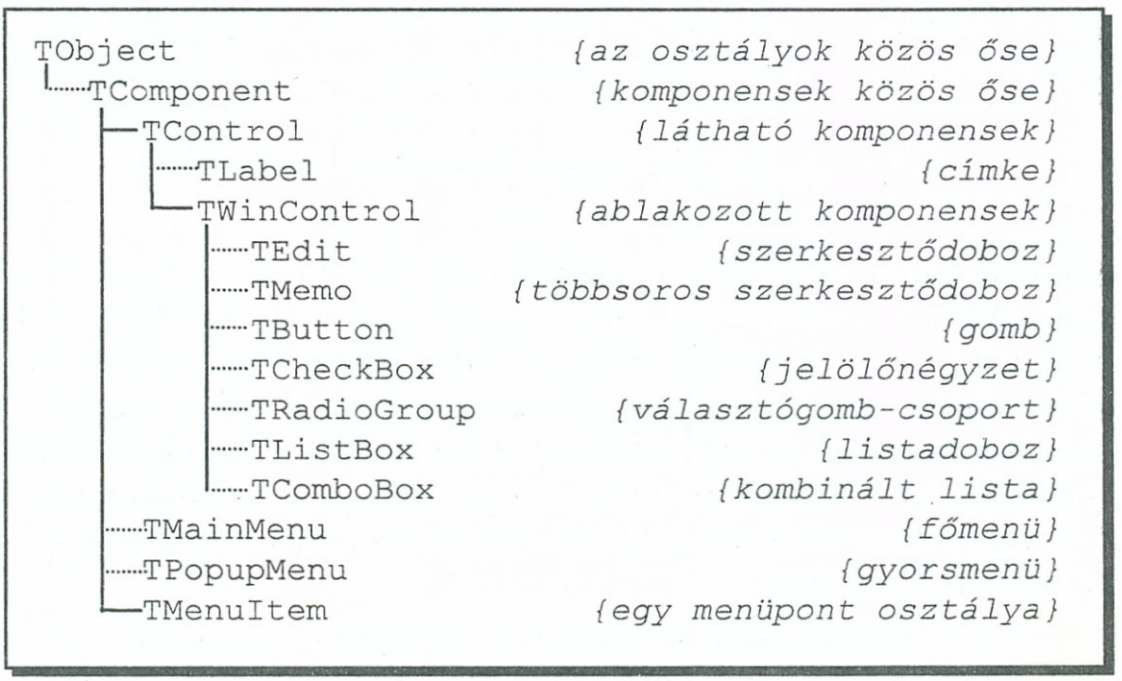
- {$H+}: a *String* típus *AnsiString*-ként viselkedik
- {$H-}: a *String* típus *ShortString* típus-ként viselkedik.

A bizonytalanságot elkerülendő használjuk az új karakterlánctípusokat: ha rövid karakterláncra van szükségünk, akkor ezeket *ShortString*-ként deklaráljuk, ha viszont hosszabbakra, akkor a *AnsiString* és *WideString* típusokat részesítsük előnyben.

4. Delphi standard komponensek

Delphiben a komponenspaletta első oldalára kerültek a leggyakrabban használt vezérlőelemek, más néven standard komponensek. Ilyen a gomb, a szerkesztődoboz, a választógomb, a jelölőnégyzet... Osztályhierarchiájukat a 4.1. ábra szemlélteti. Ebben a fejezetben megismerkedhetünk szerepükkel, használatukkal, fontosabb jellemzőikkel és metódusaikkal. Természetesen nem tarthatjuk észben jellemzőiknek és metódusaiknak sokaságát, célunk csupán az, hogy egy általános képet alkossunk róluk. A következő fejezetekben alapfogalmakként fogjuk az itt leírtakat használni, olyankor a kedves Olvasó visszalapozhat ide, és tisztázhatja felmerülő problémáit.

A 4.1.→4.13. pontok áttekintését a Delphivel most ismerkedő Olvasóimnak ajánlom. A gyakorlottabb Olvasók a 4.14. pont feladatain keresztül ellenőrizhetik le ismereteiket. A 4.15. pontban a vonszolás (*drag&drop*) technikával, a 4.16. pontban pedig az egyéni kurzorok felhasználásának módjával ismerkedhetünk meg.



4.1. ábra. Standard komponensek a Delphi osztályhierarchiában

4.1 TComponent

A *TObject* közös őse minden Delphiben fejlesztett osztálynak. Ebből származik a ***TComponent*** osztály is, mely közös őse az összes tervezési időben kezelhető komponensnek, azaz a komponens-paletták „lakóinak". Ezeket vezérlőelemeknek is szoktuk nevezni. A *TComponent* osztálytól minden vezérlőelem két fontos jellemzőt (*property*) örököl:

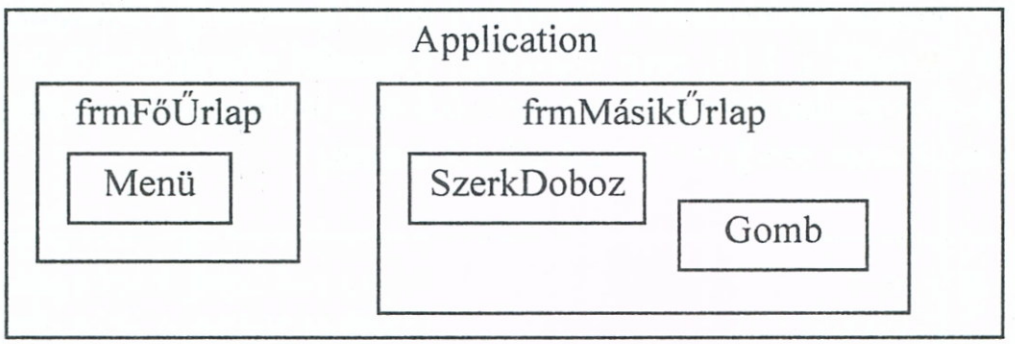
- **Name**: a komponens programbeli neve. Csak tervezési időben módosítható.

  Ha például egy gomb neve *Button1,* akkor a programban *Button1* jelenti a gomb-objektumot, *Button1.Caption* a feliratát (*Button1.Caption:= 'Ez a gomb felirata'*)...

- **Owner**: az adott komponens tulajdonosa

  Delphiben minden komponensnek van egy tulajdonosa. A tulajdonos feladata az, hogy megszűnésekor megszüntesse a tulajdonában levő elemeket is. Például egy űrlap bezárása maga után vonja a rajta lévő gombok, szerkesztődobozok... eltüntetését is.

Minden élő komponens-objektum megtalálható valahol a tulajdonosi hierarchiában. Az alkalmazásnak van egy főűrlapja, ennek tulajdonosa az alkalmazás. Az űrlapon gombok, szerkesztődobozok láthatók, tehát ezek tulajdonosa az űrlap lesz. Ha az alkalmazást bezárjuk, akkor minden űrlapjának el kell tűnnie, a rajtuk levő vezérlőelemekkel együtt. Ha viszont csak egy akármilyen (nem fő-) űrlapot zárunk be, akkor csak a saját elemeinek kell eltűnniük, maga az alkalmazás tovább él.



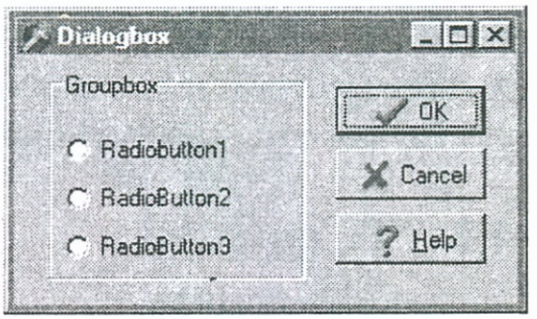
4.2. ábra. Egy alkalmazás tulajdonosi hierarchiája

A tulajdonost már a komponens létrehozásakor meg kell adnunk. Ha ez tervezési időben következik be, akkor a rendszer – a „saját kis agya szerint" – kitalálja a tulajdonost: az űrlapra elhelyezett vezérlőelemeknek maga az űrlap lesz a tulajdonosuk. Ha viszont mi programból szeretnénk egy komponens-példányt létrehozni, akkor explicit módon meg kell hívnunk a konstruktorát, és paraméterként át kell adnunk a tulajdonost (tulajdonosként legtöbbször azt az űrlapot állítjuk be, amin a komponenst meg szeretnénk jeleníteni).
Tekintsünk át egy példát: minden egérgomb lenyomásra jelenítsünk meg az űrlapon egy gombot a kattintás pozíciójában.

```
Procedure TForm1.FormMouseDown(Sender: TObject; Button:
            TMouseButton; Shift: TShiftState; X, Y: Integer);
Var
```

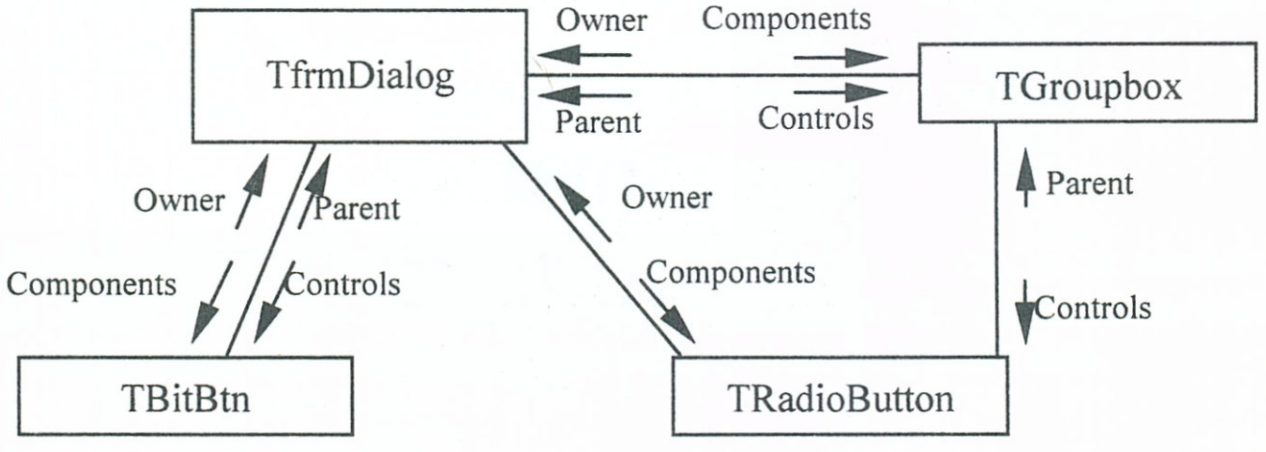
```
  Gomb: TButton;
Begin
  Gomb := TButton.Create(Self);
  Gomb.Parent := Self;
  Gomb.Caption := 'Most hoztuk létre';
  Gomb.Left := X;
  Gomb.Top := Y;
  Gomb.Width := 150;
End;
```

Van itt egy másik érdekesség is: a ***Parent*** jellemző. Minden vezérlőelemnek két „felettese" van: egy tulajdonosa (*Owner*) és egy szülője (*Parent*). Egy komponens csak mindkét felettesének beállítása után válik használhatóvá.
A tulajdonosi illetve a szülő-gyerek kapcsolatok megértésére tekintsünk át egy példát (4.3. ábra). Az ábrán egy párbeszédablak látható. Rajta egy *Ok* egy *Cancel* és egy *Help* gomb található, valamint egy választógomb-csoport (*TGroupBox* típusú) három választógombbal. Elemezzük a vezérlőelemek tulajdonosát és szülőjét:



4.3. ábra. A tulajdonosi és szülő-gyerek kapcsolat összehasonlítása

- Tulajdonos:
  - Minden vezérlőelem tulajdonosa a párbeszédablak. A tulajdonos felelős a tulajdonában levő elemek létrehozásáért, majd később a megszüntetésükért.
- Szülő:
  - A gombok és a csoport szülője a párbeszédablak.
  - A választógombok szülője a csoport. A választógomboknak van tehát egy „kisebb főnökük" is, aki – jelen esetben – azért felelős, hogy a pötty mindig csak az egyik választásnál lehessen.



4.4. ábra. A vezérlőelemek és a párbeszédablak közötti kapcsolat

Amint a 4.4. ábrán láthatjuk, a szülő–gyerek és a tulajdonos–tulajdon közötti kapcsolat kétirányú. Nemcsak a gyerek tud szülőjéről és tulajdonosáról, hanem a szülő is ismeri a gyerekeit, a tulajdonos is ismeri a tulajdonában levő elemeket. Térjünk vissza az előző párbeszédablakhoz: a párbeszédablak a *Controls* listában tárolja a gyerekeit (a választógomb-csoportot, az *OK*, *Cancel* és *Help* gombokat), míg a tulajdonában levő elemeket a *Components*-ben tartja nyilván (a választógomb-csoportot, a választógombokat és a sima gombokat). A *Tab* billentyű folyamatos leütésekor a gyerek-komponenseket, azaz a *Controls* listát járjuk körbe-körbe.

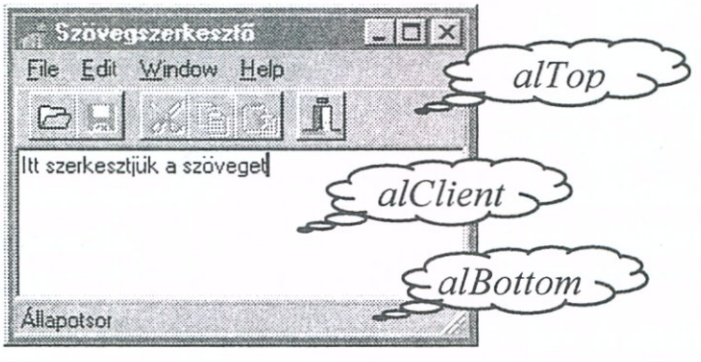
A szülő-gyerek kapcsolat két „ablakozott" vezérlőelem között értelmezett. Szabályok:

- A gyerek sohasem léphet ki a szülő területéről.
- A gyerek pozícióját a szülő bal felső sarkától mérjük.

A *Parent* jellemző a *TControl* osztályban jelenik meg, ettől kezdve ezt minden leszármazott komponensnél kötelező megadnunk (akárcsak a tulajdonost).

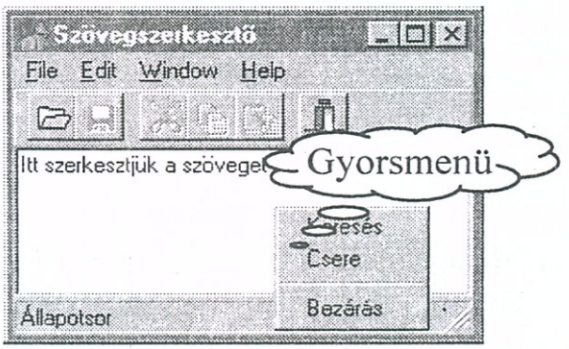
4.2 TControl

- Helye az osztályhierarchiában: *TObject/TComponent.*
- Szerepe: a Delphi látható komponenseinek (kontrolok, vezérlőelemek) közös absztrakt őse
- Fontosabb jellemzői:
  - **Align**: igazítás az űrlapon (pontosabban a szülőkomponensen) belül. Ez egy nagyon hasznos tulajdonság, hiszen segítségével el tudjuk érni például azt, hogy az állapotsor mindig az űrlap alján legyen, átméretezés után is. Lehetséges értékei:
    - *alNone*: nincs igazítás
    - *alTop, alBottom, alLeft, alRight*: igazítás az űrlap (a szülőelem) felső, alsó, bal, illetve jobb széléhez
    - *alClient*: kitöltő igazítás („kitölti a maradékot")



4.5. ábra. Példa a vezérlőelemek igazítására

- **Visible**: láthatóság állítása (*True* ⇒ látható)
- **Enabled**: fogadja-e az üzeneteket?
- **Top, Left, Width, Height**: pozíció állítása (a szülő bal felső sarkától mért értékek pixelekben)
- **Color**: a vezérlőelem színe. Lehetséges értékei: *clRed, clSilver* ... (konstansok).
- **DragCursor**: az egérkurzor vonszolás (*drag&drop*) közbeni formája. Lehetséges értékei: *crSize, crHourGlass...*, de saját tervezésűek is lehetnek (bővebben lásd a 4.16. pontban).
- **Font**: *TFont* típusú jellemző, mely a vezérlőelem feliratának betűformátumát tartalmazza. Állíthatók a betűszín (*Font.Color := clRed*), betűtípus (*Font.Name := 'Arial CE'*)...
- **PopupMenu**: az egér jobb gombjának lenyomására, helyben megjelenő függőleges menü (továbbá gyorsmenü).



4.6. ábra. A gyorsmenü használata

Most nézzük a *TControl* osztály fontosabb eseményjellemzőit[1] (*events*). Előtte viszont tisztázzuk az eseményjellemző fogalmát. A Delphi komponenseknek vannak kissé „rendhagyó", eljárás típusú jellemzői is. Ilyen például a gombok esetén az *OnClick*. Az objektum felügyelőben beírhatunk ide egy eljárásnevet, és azt fogjuk tapasztalni, hogy a gombra való kattintáskor ez az eljárás fog lefutni. Ha megjelenik az objektum-felügyelőben, akkor ennek egy publikált jellemzőnek kell lennie; és mivel eljárás típusú, értékként csak eljárásnevet adhatunk meg. Valójában az *OnClick* a következőképpen néz ki:

```
Type TNotifyEvent = Procedure(Sender: TObject);
...
Published Property OnClick: TNotifyEvent ...;
```

A *TNotifyEvent* egy olyan eljárástípus, melynek egy paramétere van, és ez általában az eseményt okozó objektum. Ha például *Gomb1*-re kattintunk, akkor meghívódik a *Gomb1.OnClick*-be írt eljárás, paraméterében pedig a *Gomb1* objektum lesz.

---

[1] Minden komponens esetén kiemelten tárgyaljuk fontosabb jellemzőit (*property*), metódusait (*methods*) és eseményjellemzőit (*events*).

Természetesen vannak más típusú eseményjellemzők is. Például az *OnMouseDown* paramétereiben egyéb információkat is átvesz (melyik egérgombot nyomta le a felhasználó, hol nyomta le...).

```
Type TMouseEvent = Procedure (Sender: TObject; Button:
          TMouseButton; Shift: TShiftState; X, Y: Integer) ;
...
Property OnMouseDown: TMouseEvent;
```

Az eseményjellemzők olyan eljárás típusú jellemzők, melyek egy adott esemény bekövetkezésekor automatikusan meghívódnak.

- Vegyük sorba a *TControl* osztály fontosabb eseményjellemzőit (*events*):
  - **OnClick, OnDblClick**: az egér egyszeri illetve kétszeri kattintására hívódnak meg

```
{Például a btnKilep gombra való kattintáskor bezárjuk az főűr-
lapot (frmMain:TfrmMain)}
procedure TfrmMain.btnKilepClick(Sender: TObject);
begin
  Close;
end;
```

  - **OnDragDrop, OnDragOver, OnEndDrag, OnStartDrag**: vonszolás közben bekövetkező események (lásd a 4.15. pontban)
  - **OnMouseDown, OnMouseUp, OnMouseMove**: az egérgomb lenyomása, felengedése, illetve az egér mozgatásakor következnek be

```
{Az egér gombjainak lenyomásakor megjelenítünk egy üzenetet.}
procedure TfrmEger.FormMouseDown(Sender: TObject;
    Button:TMouseButton; Shift:TShiftState; X, Y:Integer);
Var EgerGomb:String;
begin
  Case Button Of
    mbLeft: EgerGomb:= 'bal';
    mbMiddle: EgerGomb:= 'középső';
    mbRight: EgerGomb:= 'jobb';
  End;
  ShowMessage(Format('A (%d,%d) pozícióban nyomták le '
                + 'az egér %s gombját ', [X, Y, EgerGomb]));
end;
```

Az *OnClick,* valamint az *OnMouseDown* és *OnMouseUp* eseményjellemzők között több különbség is van. Amikor egy gomb felett kattintunk, lefutnak az *OnMouseDown*, majd az *OnMouseUp* eseményjellemzőbe épített utasítások, és természetesen az *OnClick*-be írtak is. Ha viszont nem ugyanazon elem felett engedjük fel az egér gombját, akkor az *OnClick* elmarad.

A kattintás és az egérgomb lenyomás/felengedés eseményjellemzők paraméterei is különböznek: például a *OnMouseDown*-ban megkapjuk paraméterként az egérgomb lenyomásának pozícióját, az *OnClick*-ben viszont nem. Lesznek feladatok, amelyekben ez is egy fontos szempontnak bizonyul.

4.3 TLabel (Címke)

- Helye az osztályhierarchiában: *TObject/TComponent/TControl/...TLabel*
- Szerepe: „passzív", nem szerkeszthető szövegek megjelenítése
- Fontosabb jellemzői:
  - **Caption**: maga a szöveg
  - **Transparent**: ha Igazra állítjuk, akkor a címke háttere átlátszóvá válik. Ez akkor hasznos, ha egy képet feliratozunk vele.

4.4 TWinControl

- Helye az osztályhierarchiában: *TObject/TComponent/TControl/TWinControl*

  „Ablakozott vezérlőelemek"-nek nevezzük azokat a komponenseket, melyek képesek üzenetekre reagálni, azaz rendelkeznek saját ablakfüggvénnyel. Ilyen például a gomb (*TButton*), a szerkesztődoboz (*TEdit*), a jelölőnégyzet (*TCheckBox*), a listadoboz (*TListBox*)... Mindezek közös őse a *TWinControl* osztály. Itt kerül az öröklési láncba a *MainWndProc* nevű ablakfüggvény, melyről az 1. fejezetben már beszéltünk.

- A *TWinControl* osztály fontosabb metódusai:
  - **Update**: frissítést, újrarajzolást idéz elő
  - **SetFocus**: hatására a fókusz erre a vezérlőelemre kerül át. Például *Edit1.SetFocus*
- Fontosabb eseményjellemzői:
  - **OnEnter**, **OnExit**: a vezérlőelembe való belépéskor (fókuszba kerüléskor), illetve kilépéskor (a fókusz elvesztésekor) következnek be
  - **OnKeyDown**, **OnKeyUp**, **OnKeyPress**: a billentyűk lenyomásakor, illetve felengedésekor következnek be

```
Procedure OnKeyDown(Sender: TObject; var Key: Word;
                    Shift: TShiftState);
Procedure OnKeyUp(Sender: TObject; var Key: Word;
                  Shift: TShiftState);
procedure OnKeyPress(Sender: TObject; var Key: Char);
```

A *Key* paraméter az első két metódusban a leütött billentyű virtuális kódját, míg a harmadik metódusban a billentyű ASCII kódját tartalmazza. Ebből egyenesen következik a használhatóságuk: az *OnKeyPress* metódus csak az „egyes kódú" billentyűkre (kis és nagy betűk, számjegyek és írásjelek) hívódik meg, míg az *OnKeyDown* és *OnKeyUp* a „dupla kódú" billentyűkre **is** (funkcióbillentyűk, nyílbillentyűk és különböző *Shift, Ctrl* és *Alt* billentyűkombinációk).

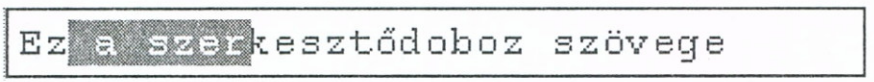
Az *OnKeyDown, OnKeyUp* és *OnKeyPress* metódusok a lenyomott billentyű kódját változó paraméterként (cím szerint) veszik át, tehát értékét meg is tudják változtatni. Ha például meg szeretnénk szűrni a billentyűzetről érkező karaktereket – például csak az 'a' betűt akarjuk átengedni –, akkor a karaktert minden egyéb esetben le kell nullázunk:

```
Procedure TForm1.Edit1KeyPress(Sender: TObject;
                                    var Key: Char);
Begin
   If Key <> 'a' Then
      Key:= #0;
End;
```

4.5 TEdit (Szerkesztődoboz)

- Helye az osztályhierarchiában: *TObject/TComponent/TControl/TWinControl/ ...TEdit*
- Szerepe: egysoros szöveges információ megjelenítése és szerkesztése
- Fontosabb jellemzői:
  - **Text**: a szerkesztődoboz szövege (Delphi 1-ben maximum 255 karaktert tartalmazhat, az újabb verziókban nincs határ. Ez az új, *AnsiString* karakterlánc típus miatt van így, lásd 3. fejezet).
  - **MaxLength**: a maximálisan megengedett karakterek száma
  - **SelText**: a kijelölt szöveg
  - **SelStart, SelLength**: a kijelölés kezdete és hossza

Például:

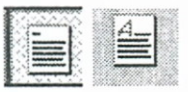


SelText = ' a szer'
SelStart = 2
SelLength = 7

A fenti jellemzők mindegyike mind tervezési, mind futási időben állítható.

- Fontosabb metódusai:
  - **SelectAll**: kijelöli a teljes szöveget
  - **ClearSelection**: törli a kijelölt szövegrészt
  - **CutToClipboard**, **CopyToClipboard**, **PasteFromClipboard**: vágólapra vágja, másolja ki a kijelölt szöveget, illetve vágólapról illeszt be
- Fontosabb eseményjellemzői:
  - **OnChange**: bekövetkezik valahányszor megváltozik a szöveg. Figyelem, ez az esemény nem azonos az *OnKeyDown*-al. Míg a jobbra-, balra billentyűk nyomogatásakor az *OnChange* nem hívódik meg, addig az *OnKeyDown* igen. Az *OnKeyPress* eseménytől pedig abban különbözik, hogy míg az *OnKeyPress* meghívásakor a szöveg még a régi (paraméterében pedig az új leütött karakter), addig az *OnChange* hívásakor a szöveg már tartalmazza a frissen beütött karaktert. Éppen emiatt, az *OnChange* nem használható a billentyűzetről érkező karakterek szűrésére (mire meghívódik, már „késő bánat").

4.6 TMemo (többsoros szerkesztődoboz), TRichEdit

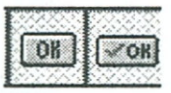


- Helyük az osztályhierarchiában: *TObject/TComponent/TControl/TWinControl/ ...TMemo, TRichEdit*. A *TRichEdit* csak a Delphi 32 bites verzióiban létezik.
- Szerepük: e két komponens többsoros szövegek szerkesztésére használható. A *Text* tulajdonságuk tárolja a teljes szöveget (Delphi 1-ben csak az első 255 karaktert). Ha soronként szeretnénk a szöveget feldolgozni, akkor erre a ***Lines*** jellemzőt használjuk.

```
Var Memo: TMemo; ...
Memo.Text → a teljes szöveg (vagy első 255 karaktere)
Memo.Lines[0] → első sor
Memo.Lines[1] → második sor
...
Memo.Lines[Memo.Lines.Count -1] → utolsó sor
```

A *TMemo* komponensben mindig csak egy betűtípus, stílus... egyféle formátum lehetséges, és ez az egész szövegre érvényes.
A Delphi 2-ben bevezetett *TRichEdit* komponens mindent „tud", amit a *TMemo*, de szövegformázási lehetőségekkel dúsítva. Amint a neve is mondja, használható benne minden *Rich Text* formátum által megengedett formázás (karakter, bekezdés, tabulátorok). Sőt, az így megszerkesztett szöveget ki is lehet menteni állományba, illetve vissza lehet állományból tölteni (*.RTF).

4.7 TButton (gomb), TBitBtn (Additional paletta)

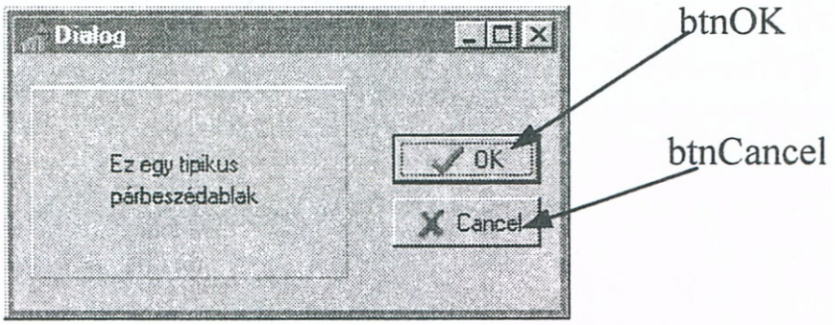


- Helyük az osztályhierarchiában: *TObject/TComponent/TControl/TWinControl/ ...TButton/TBitBtn*
- Szerepük: kattintással indított tevékenység elvégzése
- Fontosabb jellemzőik:
  - **Caption**: a gomb felirata
  - **Cancel**: ha Igazra állítjuk, akkor a gombra épített tevékenység az *Esc* billentyű hatására is meghívódik
  - **Default**: ha Igazra állítjuk, akkor a gombra épített tevékenység az *Enter*-re is meghívódik, ez lesz az „alapértelmezett" gomb.

    A *Cancel* és *Default* jellemzők beállításával a gomb nem sajátítja ki teljesen magának az *Escape* és *Enter* billentyűket. Ezek csak akkor fogják a gombra épített tevékenységsorozatot elindítani, ha a fókuszban levő vezérlőelem nem tudja lekezelni az illető billentyűt. Például, ha egy másik gombon ütjük le az *Enter*-t, akkor természetesen annak az *OnClick*-je fog lefutni, nem pedig az alapértelmezett gombé.

    A *Default* és *Cancel* jellemzőket párbeszédablakokban szoktuk használni. Ha a felhasználó *Enter*-t üt, akkor alapértelmezés szerint az *btnOK* hívódjon meg, míg ha *Esc*-et, akkor a *btnCancel*-ra írt tevékenységsorozat hajtódjon végre.

```
btnOK.Default := True;
btnCancel.Cancel := True;
```



4.7. ábra. Egy tipikus párbeszédablak

  - **ModalResult**: *mrNone, mrOK, mrCancel ...*

    A párbeszédablakok általában modálisak, azaz bezárásukig uralják az alkalmazást. Ha egy űrlapot modálisan nyitunk meg, akkor mindaddig ő lesz a fókuszban, míg el nem tüntetjük. Tegyük fel, hogy űrlapunkon van egy *OK* és egy *Cancel gomb* (mint általában a párbeszédablakokon). Mindkét gombra való kattintásra az ablaknak el kell tűnnie. Igen ám, de miután eltűnt, honnan tudjuk

meg, hogy melyik gomb miatt lett bezárva? A válasz a *ModalResult* jellemzőben rejlik.

```
btnOK.ModalResult := mrOK;
btnCancel.ModalResult := mrCancel;
```

Ha a gombok *ModalResult* jellemzőjét akár tervezéskor, akár futáskor így beállítottuk, akkor az űrlap bezárása után is lekérdezhető a bezárás oka.

```
If ParbeszedAblak.ShowModal = mrOK Then
   ShowMessage('Az OK gombra kattintottak')
Else
   ShowMessage('A Cancel gombra kattintottak');
```

Ha egy gomb *ModalResult* tulajdonságát beállítjuk *mrOk* vagy *mrCancel*-re, akkor a párbeszédablak bezárását már nem kell lekódolnunk (az *OnClick* eseményben), automatikusan ez fog történni, amikor futáskor a gombra kattintunk. A kódolás ekkor kifejezetten káros is lenne, a *ModalResult* jellemző használhatatlanságához vezetne (a fenti kódrészlet már nem működne, nem tudnánk lekérdezni a bezárást okozó gombot).

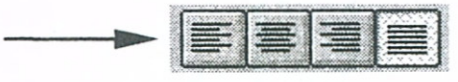
ModalResult jellemzője az űrlaposztálynak (*TForm*) is van, lásd a következő fejezetben.

- A *TBitBtn* típusú gombok nem csak feliratot, hanem különböző rajzokat is tartalmazhatnak. Ezt további három jellemzővel lehet elérni:
  - **Kind**: *bkCustom, bkOK ...*
    Beállítható egy rendszer által felkínált típus. Ha például *bkOK* típust állítunk be, akkor a tipikus „zöld pipás" gombot fogjuk látni, melynek *ModalResult* tulajdonsága is automatikusan *mrOK*-ra áll be. Ha viszont mi saját rajzolású gombokkal szeretnénk dolgozni, akkor állítsuk a *Kind* tulajdonságot *bkCustom*-ra, és a *Glyph* jellemzőbe adjuk meg a kívánt rajzot.
  - **Glyph**: *TBitmap*, **NumGlyphs**: 1..4:
    Egy gomb esetében maximum 4 bittérkép (.BMP) adható meg tervezéskor vagy futáskor. Ezeket a képeket fizikailag egy BMP állományban egymás mellett kell megadnunk a gomb normál, elszürkített, lenyomott és beragadt állapotainak megfelelően. A *TBitBtn* nem tud beragadni, így esetében csak az első három kép értelmezhető. A *TSpeedButton* komponensnél viszont már a negyediknek is van jelentősége (lásd a következő pontban).
- Fontos eseményjellemzőjük:
  - **OnClick**: ebbe írjuk a kattintásra lefutó metódus nevét

4.8 TSpeedButton (eszköztár gomb, Additional paletta)

- Helye az osztályhierarchiában: *TObject/TComponent/TControl/ ...TSpeedButton*
- Szerepe: az eszköztárakon megjelenő gombok komponense. (Eszköztárak megvalósítására nem használhatók az eddigiekben bemutatott *TButton* és *TBitBtn* komponensek, mivel ezek nem tudnak „beragadni".) A *TSpeedButton* gombokat általában egy *TPanel* komponensen helyezzük el. Többnyire grafikát tartalmaznak, de megjeleníthetünk rajtuk szöveget is. Lehetnek „beragadt", vagy „felengedett" állapotban, aktívak vagy beszürkültek. Több *TSpeedButton* példány csoportosítható; az egy csoportba tartozó gombok egymást kölcsönösen kizárják (4.8. ábra).
- Fontosabb jellemzői:
  - **Caption**: a gomb felirata
  - **Glyph**: a megjelenített kép (maximum 4 képet adhatunk meg a gomb különböző állapotainak megfelelően)
  - **GroupIndex: Integer**
    Jellemző, melynek segítségével az eszköztár gombok csoportosíthatók. Az egy csoportba tartozó gombok azonos *GroupIndex* értékkel rendelkeznek.
    Figyelem! Ha a *GroupIndex* = 0, akkor a gomb nem fog kattintáskor „beragadni", azaz *TBitBtn*-ként viselkedik. A csoportosítás csak 0-nál nagyobb *GroupIndex* értékekkel valósítható meg.
  - **Down:Boolean**
    Ha *Down=True,* akkor a gomb be van nyomva. Csak a *GroupIndex* nullától különböző értékei esetén használható.
  - **AllowAllUp:Boolean**
    E jellemző értéke meghatározza, hogy az egy csoportba tartozó gombok közül lehet-e mind egyszerre „felengedve" (*AllowAllUp=True*), vagy pedig egyiküknek kötelező módon „benyomva" kell maradnia (*AllowAllUp = False*).

Word-ben a bekezdés-igazító négy gomb egymást kölcsönösen kizárja (azonos *GroupIndex* értékkel rendelkeznek), és egyikük mindig „be van nyomva" (*AllowAllUp=False*).



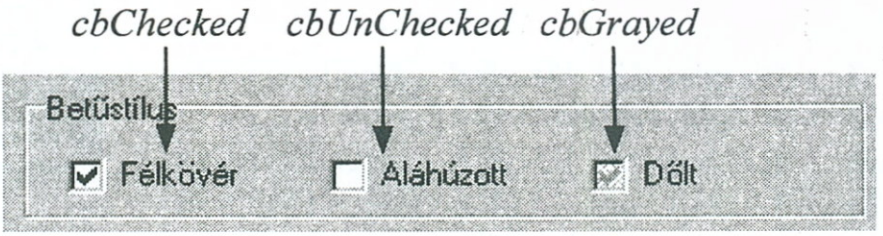
A rajzoló eszköztár gombjai is kölcsönösen kizárják egymást (egy csoportba tartoznak), de amikor nem rajzolunk, akkor egyikük sincs kiválasztva. (*AllowAllUp=True*).

4.8. ábra. Eszköztár példák

- Fontosabb metódusa:
  - **OnClick**: azonos a *TButton* és *TBitBtn* metódusával

4.9 TCheckBox (jelölőnégyzet)

- Helye az osztályhierarchiában:
  TObject/TComponent/TControl/TWinControl/ ... TCheckBox
- Szerepe: a jelölőnégyzetet általában két állapotú érték jelzésére használjuk: Igen/Nem, Igaz/Hamis, Akarom/Nem akarom, Férfi/Nő... Néha találkozhatunk szürkített jelölőnégyzettel is, ez a „nem tudom", „nem egyértelmű" vagy „mindegy" állapotot jelenti.
- Fontosabb jellemzői:
  - **Checked**: ha nincs engedélyezve a szürkítés, akkor ezzel a tulajdonsággal kérdezhetjük le a jelölőnégyzet állapotát. Igaz vagy hamis az értéke, attól függően, hogy ki van-e pipálva vagy sem.
  - **AllowGrayed**: Igazra állításával engedélyezzük a szürkítést. Ilyenkor folyamatos kattintásokra kipipáljuk, töröljük, majd szürkítjük. Most már a jelölőnégyzetnek három különböző értéke lehet, tehát a *Checked* tulajdonság már nem mond eleget. Ilyenkor a *State* jellemzőt használjuk.
  - **State**: *cbChecked, cbUnChecked, cbGrayed* (lásd 4.9. ábra).



4.9. ábra. Jelölőnégyzetek lehetséges állapotai

- Fontos eseményjellemzője:
  - **OnClick**: kattintás eseménye
    Ha azt akarjuk, hogy a változtatás azonnal érezhető legyen, akkor a jelölőnégyzet állapotát az *OnClick*-ben értékeljük ki.

4.10 TRadioGroup (választógomb-csoport)

- Helye az osztályhierarchiában:
  TObject/TComponent/TControl/TWinControl/ ...TRadioGroup
- Szerepe: egymást kölcsönösen kizáró opciók számára használatos
- Jellemzői:
  - **Items**: az opciók szövege külön sorokban

Items[0] = 'Piros', Items[1] = 'Kék' ...
Az utolsó elem indexe: *Items.Count-1*

- **ItemIndex**: a kijelölt elem indexe
  Példánkban *ItemIndex* = 2.

- Fontos eseményjellemzője:
  - **OnClick**

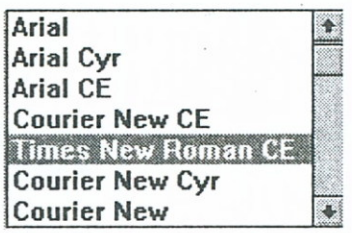
4.10. ábra. Választógomb-csoport

4.11 TListBox (listadoboz)

- Helye az osztályhierarchiában:
  TObject/TComponent/TControl/TWinControl/ ...TListBox
- Szerepe: megjelenít egy értéklistát, melyből mi egyet (esetleg többet) választhatunk. A lista elemei nem szerkeszthetők. Akkor használjuk, ha korlátozni szeretnénk a bevihető értékeket a lista elemeire. A kombinált listától (*TComboBox*) eltérő módon, a listadoboz állandóan látható; a kiválasztott elemet egy kijelölő sáv jelzi.

  Például a 4.11. ábra egy olyan listadobozt mutat be, mely tartalmazza a konkrét gépen létező betűtípusokat.

- Fontosabb jellemzői:
  - **Items**: a lista elemei (*Items[0]= 'Arial', Items[1]= 'Arial Cyr'...*)
  - **ItemIndex**: a kijelölt elem indexe. Figyelem, ez csak akkor használható, ha egy elem van kijelölve.
  - **Multiselect**: többszörös kijelölés engedélyezése (*True*), illetve tiltása (*False*). Ha egyszerre több sort is ki tudunk jelölni[1], akkor a kijelölt sorok lekérdezésére a következő két jellemzőt használjuk.



4.11. ábra. Listadoboz

  - **SelCount**: megadja a kijelölt sorok számát
  - **Selected**: egy logikai tömb, mely segítségével minden elemről sorban lekérdezhető az állapota: ki van-e jelölve vagy sem. Az alábbi programrészletben megjelenítjük a *ListBox1* listadoboz kijelölt elemeinek számát és tartalmát:

```
Procedure TForm1.Button1Click(Sender: TObject);
Var S:String; I:Integer;
Begin
With ListBox1 Do
   Begin
     S:= 'A listadobozban '+ IntToStr(SelCount) +
```

[1] A többszörös kijelölés a Windows rendszerben megszokott módon történik: a *Shift* kíséretében történő kattintással egymás melletti elemeket jelölhetünk ki, míg *Ctrl*-al egymástól távol eső elemeket „csipegethetünk össze".

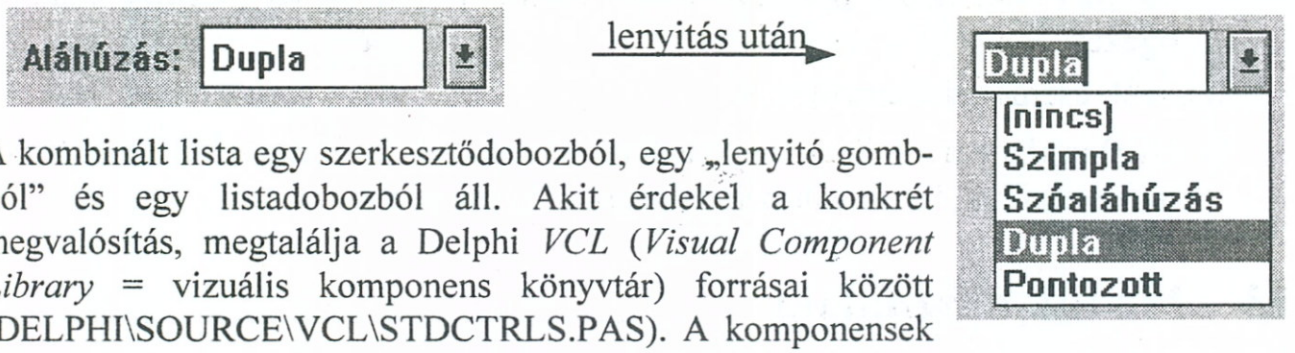
```
        ' sor van kijelölve, pontosabban a ';
      For I:=0 To Items.Count-1 Do
        If Selected[I] Then
          S:= S+ IntToStr(I) + ', ';
      {Az utolsó ', ' kitörlése}
      S:= Copy(S, 1, Length(S)-2)+ ' sorok';
      ShowMessage(S);
    End;
End;
```

- Fontos metódusa:
  - **ItemAtPos**: visszaadja a paraméterként megadott x, y koordinátákban levő sorának indexét (ha ott van egyáltalán valami). Ezt például akkor használjuk, amikor egy listadoboz elemeit át akarjuk vonszolni egy másik listadobozba. A vonszolás elkezdésének pillanatában ki kell értékelnünk a helyzetet: van-e ott valami vonszolandó vagy nincs? Magyarán: abban a pontban, ahova kattintott a felhasználó, található-e listaelem vagy sem? Ezt a vizsgálatot az *ItemAtPos* metódussal fogjuk megvalósítani (lásd 4.15. pont).

4.12 TComboBox (kombinált lista)

- Helye az osztályhierarchiában:
  TObject/TComponent/TControl/TWinControl/...TComboBox
- Szerepe: egy szöveges adat bevitelére szolgál, olyan adatéra, melyet vagy közvetlenül beírunk, vagy egy lebomló listából választunk ki. Állandó jelleggel csak a szerkesztődoboz része látható.

  Például: karakterformázásnál az aláhúzást egy kombinált listában állíthatjuk be:



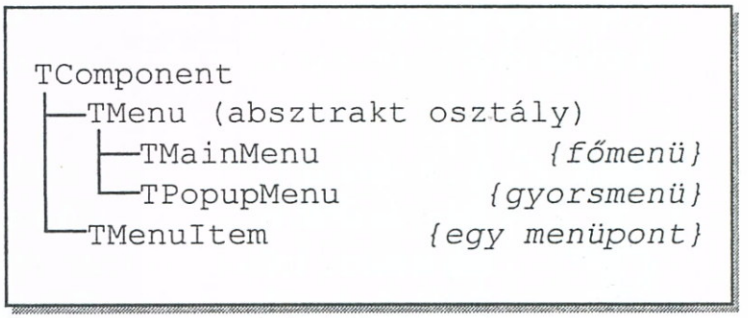
A kombinált lista egy szerkesztődobozból, egy „lenyitó gombból" és egy listadobozból áll. Akit érdekel a konkrét megvalósítás, megtalálja a Delphi *VCL* (*Visual Component Library* = vizuális komponens könyvtár) forrásai között (DELPHI\SOURCE\VCL\STDCTRLS.PAS). A komponensek forrását csak a nem standard Delphi változataihoz csatolták.

A kombinált listáknál általában be van építve egy gyorskeresési (*incremental search*) lehetőség, melynek köszönhetően a gép már a betűk folyamatos leütése közben rákeres a megegyező kezdetű listaelemekre. Delphiben ez kimaradt a *TComboBox* komponensből, de sebaj, írunk majd ezt is tudó kombinált listát a 15. fejezetben.

- Fontosabb jellemzői:
  - **Text**: a kombinált lista szövege. Példánkban *Text = 'Dupla'*
  - **Items**: a listadoboz elemei. Ha csak néhány konstans értékről van szó (mint a példánkban), akkor ezeket tervezési időben szoktuk megadni. Ha viszont a lista hosszú, és ráadásul tartalma változékony (például mindig a létező betűtípusokat kell tartalmaznia), akkor feltöltését futáskor kell megvalósítanunk, valamikor még a lista megjelenése előtt. Ha azt szeretnénk, hogy elemei automatikusan egy adatbázistáblából származzanak, akkor ne ezt a komponenst használjuk, hanem az adatbázisos párját, a *TDBLookupComboBox* komponenst (bővebben lásd a 9. fejezetben).
  - **ItemIndex**: a kiválasztott listaelem indexe (példánkban 3)
  - **Sorted**: beállítható, hogy a lista elemei növekvő sorrendben legyenek-e vagy sem
  - **SelText**, **SelStart**, **SelLength**: a kijelölt szövegre vonatkozó adatok, mint a *TEdit* komponensnél
  - **Style**: *csDropDown, csDropDownList ...*
    A kombinált lista stílusa. Ha értéke *csDropDown*, akkor a kombinált lista szerkeszthető. (Figyelem, a frissen begépelt elem nem válik automatikusan a lista részévé.) Ha viszont azt szeretnénk, hogy a felhasználó csak a lista elemei közül tudjon válogatni, ne gépelhessen be semmi mást, akkor stílusát *csDropDownList*-re állítsuk; ezzel letiltjuk a szerkesztést, kattintáskor a lista automatikusan le fog nyílni. A *Style* jellemző még a *csOwnerDrawFixed* és a *csOwnerDrawVariable* értékekkel is rendelkezhet. Ezek az egyéni rajzolású listadobozok használatát teszik lehetővé (például, ha minden eleme elé egy kis képet szeretnénk megjeleníteni). Ilyenkor az elemek rajzolását nekünk kell átvállalnunk, és ezt az *OnDrawItem* eseményében fogjuk megvalósítani (lásd 6. fejezet).
- Fontosabb eseményjellemzője:
  - **OnDropDown**: a lista lenyitásakor következik be. Ekkor szoktuk frissíteni a lista tartalmát, ha ez változik a program futása során.

4.13 Menük használata

Delphi alkalmazásainkban „játszva" készíthetünk menüket, hiszen erre is léteznek már a megfelelő komponensek. A menüknek két típusát különböztetjük meg: főmenü és gyorsmenü. Főmenünek (*MainMenu*) nevezzük az űrlapok tetején látható vízszintes menüsort, gyorsmenünek (*PopupMenu*) pedig az egér jobb gombjá-



4.12. ábra. Menükomponensek

val előhívható függőleges menüt. Delphiben mindkét típus megvalósítható a 4.12. ábra komponenseinek segítségével.
Mindkét típusú menü megvalósítható akár tervezéskor, akár futáskor, a gyakorlatban viszont ezeket tervezéskor szoktuk létrehozni, mivel ekkor egy vizuális menüszerkesztő is a rendelkezésünkre áll. Elhelyezünk tehát egy *MainMenu* vagy *PopupMenu* komponenst az űrlapon (a megvalósítandó menütípustól függően), majd behívjuk a menüszerkesztő ablakot (ehhez duplán kell kattintanunk a komponens felett). A menüszerkesztőben sorban begépeljük a menüpontokat, beállítjuk tulajdonságaikat.
Futáskor lesznek nem elérhető menüpontok is. Ezeket alkalomadtán vagy leszürkítjük, vagy mindenestől eltüntetjük. Ha szürkítéssel jelezzük egy menüpont inaktivitását, akkor a menü váza állandó lesz a program teljes futása alatt. Ha viszont dinamikusan akarunk menüpontokat hozzáadni és eltüntetni, akkor tervezéskor csak a vázát szerkesszük meg (az állandóan látható menüpontokkal), a többit pedig hagyjuk futási időre.

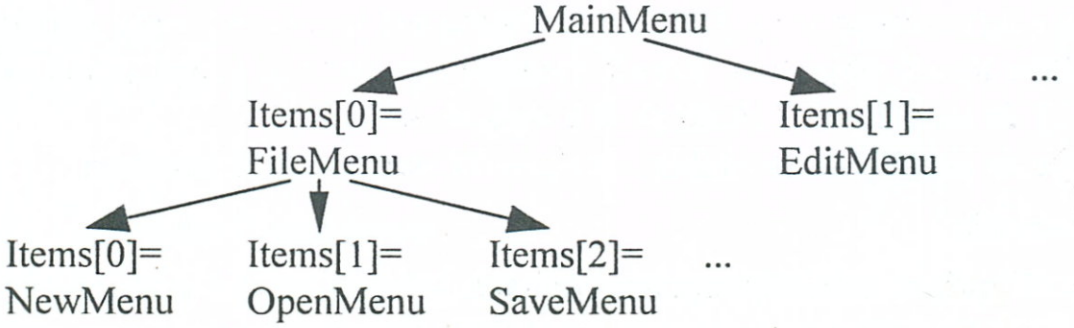
4.13.1 TMainMenu (Főmenü)

- Helye az osztályhierarchiában: *TObject/TComponent/ ...TMainMenu*
- Fontosabb jellemzői:
  - **Items**: a főmenü menüpontjai. Egy menüpontnak további (al)menüpontjai lehetnek, amint ezt a 4.13. ábra is mutatja.

Például Hozzunk létre űrlapunkon egy *MainMenu* főmenü komponenst az alábbi szerkezettel:

File	Edit	...
New	Find	
Open	Search	
Save		...
Exit		

Mindezt a vizuális menüszerkesztővel hozzuk létre, ezzel párhuzamosan a rendszer a háttérben felépít egy többágú fa-szerkezetet.



4.13. ábra. Menüszerkezet megvalósítása Delphiben

Tehát az *Open* menüpontra hivatkozhatunk *OpenMenu*-ként, vagy *MainMenu.Items[0].Items[1]*-ként. Természetesen az elsőt választjuk, a név segítségével sokkal kényelmesebb lesz majd a programozás.

És még egy fontos info: minden menüpontnak van egy *OnClick* eseménye. A faszerkezet leveleiben található menüpontok kattintására építjük be a megfelelő tevékenységeket (*NewMenu* ⇒ új állomány létrehozása, *SaveMenu* ⇒ mentés ...), a többi menüpont kattintására pedig az almenüpontok engedélyezését, illetve letiltását szoktuk beírni. Példánkban a *FileMenu.OnClick* eseményére vizsgálhatnánk felül a *NewMenu*, *OpenMenu*, *SaveMenu* és *ExitMenu* állapotát.

4.13.2 TPopupMenu (Gyorsmenü)

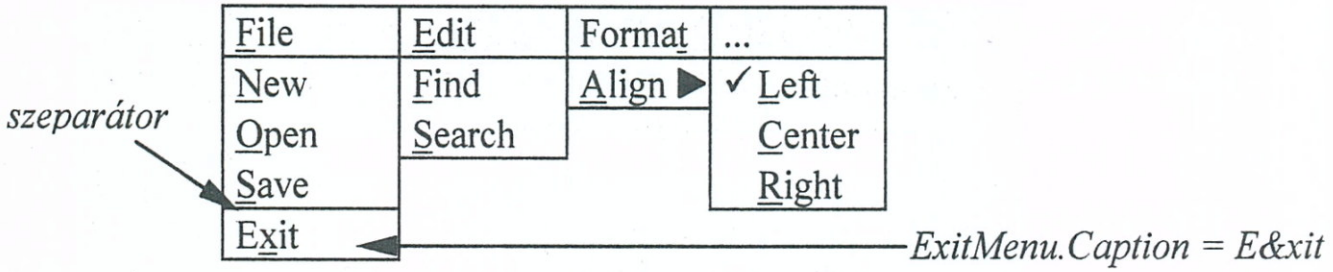
Egy űrlapon általában egy *MainMenu* komponens található, azonban *PopupMenu* komponens akár több is lehet. A gyorsmenüket más komponensek (gombok, szerkesztődobozok, listák...) *PopupMenu* tulajdonságához szoktuk kötni, azaz beállítjuk, hogy az egér jobb gombjára kattintva hol, milyen gyorsmenü jelenjen meg.

- Helye az osztályhierarchiában: *TObject/TComponent/ ...TPopupMenu*
- Fontosabb jellemzői:
  - **Items**: menüpontjai, akárcsak a főmenünél
  - **AutoPopup**: Boolean
    Általában a gyorsmenüt az egér jobb gombjával szoktuk előhívni (ez automatikusan történik, be van építve a rendszerbe). Delphiben ezt letilthatjuk (*AutoPopup:=False*), ilyenkor viszont nekünk kell valahol a programban megjelenítenünk a menüt. Ezt a ***Popup*** metódusával tehetjük meg.
- Fontosabb eseményjellemzői:
  - **OnPopup**: a menü megjelenésekor hívódik meg. Ide szoktuk beépíteni a menüpontjainak engedélyezését, illetve letiltását.

4.13.3 TMenuItem

Minden – tervezéskor vagy futáskor – létrehozott menüpont vagy almenüpont valójában *TMenuItem* típusú objektum. Ennek megfelelően vannak adatai (címe, állapota, almenüpontjai...) és metódusai. Tekintsük át a legfontosabbakat:

- Helye az osztályhierarchiában: *TObject/TComponent/TMenuItem*
- Jellemzők:
  - **Caption**: a menüpont címe, felirata.
    Például *FileMenu.Caption = '&File'*. Ez azt jelenti, hogy a menüpont felirata a *File* lesz, és az *Alt+F* lenyomására is meghívható lesz. A szeparátor (tagoló vonal) egy olyan menüpont, melynek *Caption* jellemzője a kötőjel ('-').



4.14. ábra. Egy szövegszerkesztő menüszerkezete

- **Checked**: segítségével „pipás" menüpontokat hozhatunk létre. Példánkban (menüszerkezetéből kiindulva egy szövegszerkesztőre gondolhatunk) az igazítást (*Align*) a *Left* menüpont előtti pipa jelzi. Sajnos a *Left*, *Center* és *Right* menüpontokat nem tudjuk csoportosítani, így nem automatizálható a pipa figyelése sem. Amikor középre igazítjuk szövegünket, akkor nekünk kell a programból letörölnünk a pipát a *Left* elől *(LeftMenu.Checked:= False)*, és nekünk kell ezt meg is jelenítenünk a *Center* előtt *(CenterMenu.Checked:= True)*.
- **Enabled**: a menüpontok szürkítését, letiltását teszi lehetővé
- **Items**: almenüpontjainak tömbje (ha vannak)
- **Count**: almenüpontjainak száma
- **GroupIndex**: ezzel szabályozhatjuk a menüpontok összefésülését, lásd az 5. és 18. fejezetekben.

• Metódusai:

- **Add, Insert, Remove**: új menüpontok dinamikus hozzáadását és törlését teszik lehetővé

• Eseményjellemzője:

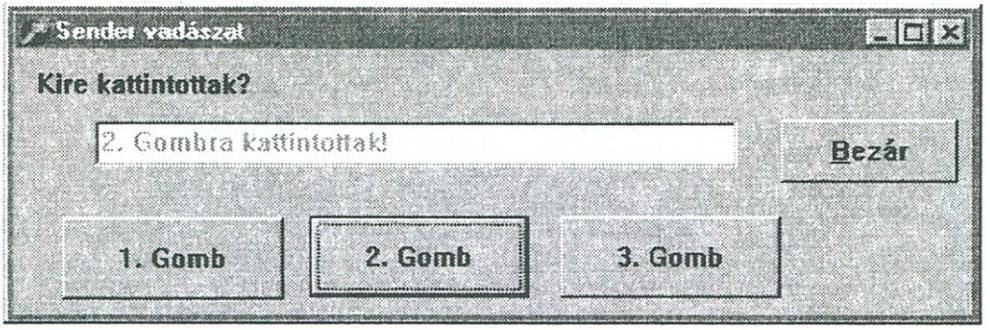
- **OnClick**

4.14 Feladatok

Immár megismerkedtünk az alapfogalmakkal, oldjunk meg együtt néhány feladatot, melyeken eddigi ismereteinket begyakorolhatjuk.

4.14.1 „Sender" vadászat

Helyezzünk el űrlapunkon három gombot. Valahányszor kattintunk a gombokra, mindannyiszor jelenjen meg a szerkesztődobozban egy szöveg a lenyomott gomb feliratával. Körülbelül így:



4.15. ábra. „*Sender*" vadászat

Megoldás (🖫 4_SENDER\SENDERP.DPR)

Tervezzük meg űrlapunkat! Helyezzük el rajta a szükséges komponenseket, állítsuk be jellemzőiket:

Komponens	Jellemző	Érték
eEredmeny: TEdit	Text	" *{az elején még üres}*
	Enabled	False
btnGomb1: TButton	Caption	1. Gomb
btnGomb2: TButton	Caption	2. Gomb
btnGomb3 : TButton	Caption	3. Gomb
btnBezar: TButton	Caption	&Bezár

A különböző gomboknak ugyanazt a feladatot kell ellátniuk: meg kell jeleníteniük egy információs szöveget a szerkesztődobozban. Jó lenne tehát, ha nem kellene három eseménykezelő metódust írnunk, csak egyet (*GombClick*), és ezt hívná meg minden gomb[1].

```
btnGomb1.OnClick := GombClick
btnGomb2.OnClick := GombClick
btnGomb3.OnClick := GombClick
```

Igen ám, de a *GombClick* eljárásban tudnunk kellene, hogy konkrétan melyik gombra kattintottunk, melyik gomb miatt hívódott meg. Erre való a *Sender* paraméter: minden eseménykezelő kap legalább egy paramétert (a *Sender*-t), mely az eseményt okozó objektum (kevés kivétellel: például a *drag&drop* technikában a *Sender* mást jelent, lásd 4.15. pontban). Így egyértelműen el tudjuk dönteni, hogy az eseményt melyik gomb kezdeményezte. A *Sender* paraméter *TObject* típusú, hiszen bárki lehet egy esemény okozója: egy gomb, egy szerkesztődoboz, egy menüpont... Alkalomadtán rákérdezhetünk konkrét típu-

[1] Ha mi szeretnénk elnevezni az eseménykezelő metódust (nem szeretjük a rendszer által generált nevet), akkor az objektum-felügyelőben a kívánt esemény mellé előbb beírjuk az új metódus nevét, majd utána *Enter*-t ütünk. Példánkban a *GombClick* legyen ez a név. Az első gombnál begépeljük és kifejtjük, majd a többinél már a listából választjuk ki.

sára, sőt típuskényszerítést is végezhetünk. De térjünk vissza a példánkhoz: a *TObject*-ben még nincs *Caption* jellemző, a *TButton*-nál viszont már van; mivel mi ezt a jellemzőt szeretnénk használni, típus-átalakítást kell alkalmaznunk.

```
procedure TfrmSender.GombClick(Sender: TObject);
begin
   If Sender is TButton Then
      With Sender As TButton Do
         eEredmeny.Text := Caption +  'ra kattintottak!';
end;
```

Így mindhárom gomb ugyanezt az eseménykezelőt hívja, a szerkesztődoboz szövege mégis mindig a megfelelő gomb feliratát tartalmazza.

Mindig „szépen" programozzunk! Előbb gondoljuk végig a feladatot, és a legrövidebb, leghatékonyabb megoldást válasszuk. Ne töltsük meg forráskódunkat sok, fölösleges „nyúlfarknyi" eljárással.

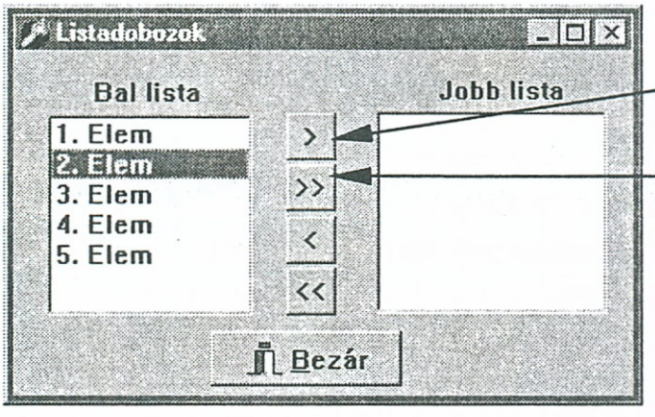
A *btnBezar* gomb *OnClick* metódusába írjuk a következőket:

```
procedure TfrmSender.btnBezarClick(Sender: TObject);
begin
   Close;
end;
```

A *TForm* osztály *Close* metódusát hívjuk meg, hatására pedig űrlapunk (*TfrmSender* példánya) bezárul. S mivel ez egyben alkalmazásunk főablaka is, az alkalmazásnak is vége.

4.14.2 Listadobozok

Következzen most egy kicsit bonyolultabb feladat: legyen űrlapunkon két listadoboz. Ezek elemeit lehessen áthelyezni balról-jobbra, illetve jobbról-balra a megfelelő gombok segítségével.



4.16. ábra. Listadobozok

Megoldás (🖫 4_LISTAK\LISTAP.DPR)

Tervezzük meg űrlapunkat. Nevezzük el *frmListaDobozok*-nak, majd helyezzük el rajta komponenseinket.

Komponens	Jellemző	Érték
lbBal, lbJobb: TListBox	Items	"
	MultiSelect	True
btnJobbra: TButton	Caption	>
btnJobbraMind: TButton	Caption	>>
btnBalra: TButton	Caption	<
btnBalraMind: TButton	Caption	<<
btnBezar: TBitBtn	Caption	&Bezár
	Kind	bkClose

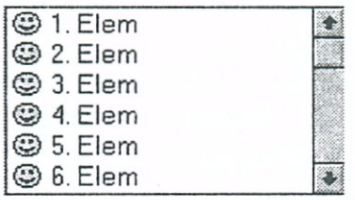
Kezdetben a két listadoboz üres. Tervezési időben is feltölthetnénk értékekkel (az objektum-felügyelőben az *Items* jellemző segítségével), azonban most mi futási időben fogjuk ezt megtenni (így többet tanulhatunk a feladatból). Tehát még mielőtt megjelennének a listák, fel kell ezeket töltenünk. Melyik legyen ez az esemény? Az *frmListaDobozok OnCreate* eseménye az űrlap létrehozásakor hívódik meg. Ide fogjuk beépíteni a *lbBal* feltöltését (a *lbJobb*-ot üresen hagyjuk). Igen ám, de hogyan lehet programból feltölteni az *Items* jellemzőt?

A listadobozok *Items* jellemzője ***TStrings*** típusú. Ez egy olyan osztály, mely többsoros szöveges információk tárolását és karbantartását teszi lehetővé. Elhelyezhetünk benne szövegeket külön sorokban (minden sor maximum 255 karakterből állhat), törölni tudunk belőle, sorait fel tudjuk cserélni stb., sőt a szövegeket még állományba is lementhetjük, illetve onnan betölthetjük. A következőkben a *TStrings* osztály fontosabb jellemzőivel és metódusaival fogunk megismerkedni:

A TStrings osztály

- Jellemzők:
  - **Count**: a tárolt adatok száma (sorok száma)
  - **Strings**: a karakterláncok tömbje
    Strings[0] = első sor, *Strings[1]* = második sor... *Strings[Count-1]* utolsó sor
  - **Objects**:

Az *Objects* és a *Strings* jellemzőket két párhuzamos tömbként foghatjuk fel. A *Strings*-ben szövegeket, az *Objects*-ben pedig egyéb objektumokat tárolhatunk, például bittérképeket. Minden sornak megfeleltethetünk egy rajzot, melyet a listadobozban a szöveg előtt jelenítünk majd meg (4.17. ábra[1]).



4.17. ábra. Egy „vidám" listadoboz

- Metódusok:
  - **Add, Insert**: új elemek hozzáadását teszik lehetővé: az új elemet az *Add* utolsó sorban helyezi el, az *Insert* pedig a megadott pozícióba illeszti be. Az első sor indexe nulla.
    Például:

```
ListBox1.Items.Add('ez lesz az utolsó sor');
ListBox1.Items.Insert (1, 'ez a második sor lesz');
```

  - **Delete**: törli az adott indexű elemet
  - **Clear**: töröl minden elemet
  - **Exchange**: két megadott indexű elemet felcserél
  - **LoadFromFile, SaveToFile**: állományból tölti be, illetve menti ki a listadoboz tartalmát (a szöveges állomány sorainak száma megegyezik a listadoboz sorainak számával.)
    Például

```
ListBox1.Items.LoadFromFile('SZOVEG.TXT');
```

Térjünk vissza a listadobozos feladatunkhoz! Az *lbBal* lista feltöltését a következőképpen valósítjuk meg:

```
procedure TfrmListadobozok.FormCreate(Sender: TObject);
Var I:Integer;
begin
  With lbBal.Items Do
    For I:= 1 To 5 Do
      Add(IntToStr(I)+'. Elem');
  GombAllitas; {erről később beszélünk}
end;
```

Mit írjunk a gombok *OnClick* eseményére? A *btnJobbra* és a *btnBalra* gombok ugyanazt teszik – a kijelölt elemeket helyezik át –, csak más irányba: balról-jobbra illetve jobbról-balra. Írjunk tehát egy közös metódust (*KijeloltetClick*), és ezt hívjuk meg mindkét gomb esetén:

[1] Konkrét megvalósítását lásd a 6. fejezetben.

```
procedure TfrmListadobozok.KijeloltetClick(Sender: TObject);
begin
  {Ha a Jobbra gombra kattintottunk, akkor}
  If Sender = btnJobbra Then
    KijeloltetViszi(lbBal, lbJobb)
  Else
    KijeloltetViszi(lbJobb, lbBal);
end;
```

A *KijeloltetViszi* metódust mi vezetjük be, tehát ezt deklarálnunk kell a *TfrmListaDobozok* osztályban. Mivel csak a fenti metódusból fogjuk hívni, elég, ha privátként deklaráljuk. Paraméterként átvesz két listadobozt (tulajdonképpen két listadobozra mutatót!), a *Forras* és *Cel* dobozokat. Feladata, hogy a *Forras* lista kijelölt elemeit áthelyezze a *Cel* listába.

```
type
  TfrmListadobozok = class(TForm)
    lbBal: TListBox;
    ...
  private
    Procedure KijeloltetViszi(Forras, Cel:TListBox);
    ...
  end;
...
Procedure TfrmListadobozok.KijeloltetViszi (Forras, Cel: TListBox);
Var I:Integer;
Begin
  For I:=Forras.Items.Count-1 DownTo 0 Do
    If Forras.Selected[I] Then
      Begin
        Cel.Items.Add(Forras.Items[I]);
        Forras.Items.Delete(I);
      End;
  GombAllitas;
End;
```

Két érdekesség is van ebben a kódrészletben:

- Miért haladunk visszafelé a ciklusban, az utolsó elemtől az elsőig? A válasz egyszerű: tegyük fel, hogy a *Forras* listában összesen öt elem van, és ezek közül az első van kijelölve. Ha az első elemtől indulnánk (0→4), tapasztalnánk, hogy azt máris át kell dobnunk a másik listába; tehát áthelyezzük, majd kitöröljük a *Forras*-ból. Igen ám, de ettől a pillanattól kezdve listánknak már csak négy eleme van (0→3), tehát nem lenne szabad a 4. indexre rákérdeznünk, hivatkoznunk. Ha viszont így írjuk meg a ciklust (0→4), akkor a ciklusváltozó felveszi a 4 értéket is! Ekkor programunk futási hibával leáll. A fordított irányú feldolgozás megoldja ezt a problémát.
- A másik érdekesség az utolsó utasításban található: *GombAllitas*. Ez a metódus minden elem-mozgatás után kiértékeli, hogy mely gomboknak kell továbbra is elérhetőknek lenniük és melyeknek nem. Így, ha például az *lbBal* üres, akkor a *btnJobbra* és a *btnJobbraMind* gomboknak sincs túl sok értelmük, tehát ezeket inaktívvá tehetjük.

```
Procedure TfrmListadobozok.GombAllitas;
Begin
  btnJobbra.Enabled := lbBal.SelCount >0;
  btnJobbraMind.Enabled:= lbBal.Items.Count >0;
  btnBalra.Enabled := lbJobb.SelCount >0;
  btnBalraMind.Enabled:= lbJobb.Items.Count >0;
End;
```

Már csak a *btnJobbraMind* és *btnBalraMind* gombok eseménykezelőit (*MindetClick*) kell megírnunk. Ennek érdekében bevezetünk még egy privát metódust, a *MindetViszi*-t.

```
procedure TfrmListadobozok.MindetClick(Sender: TObject);
begin
  If Sender = btnJobbraMind Then
    MindetViszi(lbBal, lbJobb)
  Else
    MindetViszi(lbJobb, lbBal);
end;

procedure TfrmListadobozok.MindetViszi (Forras, Cel: TListBox);
Var I:Integer;
Begin
  For I:=Forras.Items.Count-1 DownTo 0 Do
    Begin
      Cel.Items.Add(Forras.Items[I]);
      Forras.Items.Delete(I);
    End;
  GombAllitas;
End;
```

Tesztelje le az alkalmazást!

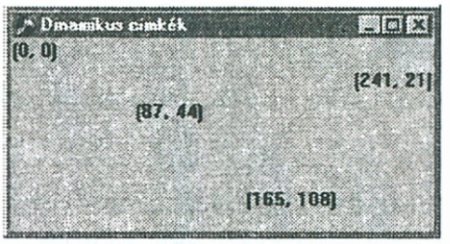
Mit kellene tenni annak érdekében, hogy az elemek mindig növekvő sorrendben helyezkedjenek el a listákban? Két megoldás is létezik: egy egyszerűbb, és egy „munkásabb".

- Első megoldás: állítsuk Igazra a listadobozok *Sorted* jellemzőjét. Így a sorrend felügyeletét a rendszerre bízzuk.
- Második megoldás: az új elemeket ne az *Add* metódussal fűzzük fel, hanem *Insert*-tel illesszük be ezeket az előzőleg általunk megkeresett pozícióba. De ki akar fölöslegesen dolgozni? Nyilvánvaló, az első megoldás győzött!

Tanulmányozza át a Delphi által felkínált *Dual List Box* űrlapmintát (*File/New/Forms/Dual listbox*)! Pont a mi feladatunkat tartalmazza.

4.14.3 Vezérlőelem létrehozása futás közben

Kezdetben az űrlapunk üres. Futás közben, minden űrlapra történő kattintáskor jelenítsünk meg egy-egy címkét a kattintás pozíciójával.



4.18. ábra. Dinamikus címkék

Megoldás (🖫 4_DINAMI\DINAMP.DPR)

Tervezési időben még nem tudjuk hány címkére lesz szükségünk. Ez attól függ milyen szaporán fog kattintgatni a felhasználó. Emiatt, a címkék létrehozását futási időre hagyjuk. Igen ám, de hova építsük be ezt a tevékenységet? Logikus az lenne, hogy ha kattintáskor kell a címkének megjelennie, akkor az *OnClick* eseménykezelőbe írjuk a létrehozását. Csakhogy ez az eseményjellemző nem rendelkezik olyan paraméterrel, amely elárulná a kattintás pozícióját. Vissza az egész, ez nem megoldás.
Vizsgáljuk meg az *OnMouseDown* eseményt: paraméterlistájában megtalálhatók az egér X, Y koordinátái, így ezt fogjuk használni.
Most már csak azt kell eldöntenünk, hogy a címkék létrehozását melyik objektum *OnMouseDown* eseménykezelőjébe írjuk? A válasz egyszerű, egyrészt azért, mert csak egy vizuális objektumunk van (maga az űrlap), másrészt pedig azért, mert rajta kattintunk.
Íme a kód:

```
procedure TfrmCimkek.FormMouseDown(Sender: TObject;
     Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var Cimke:TLabel;
begin
  Cimke:= TLabel.Create(Self);
  with Cimke Do
    Begin
      Parent := Self;
      Top := Y;
      Left := X;
      Caption := '(' + IntToStr(X) + ', ' + IntToStr(Y) + ')';
    End;
end;
```

Vajon mi történik a rengeteg létrehozott címkével? A program befejezése után is tovább foglalják a memóriát? Természetesen nem. Minden vezérlőelemnek van egy szülője és egy tulajdonosa. Ezek egy listára felfűzve tárolják az objektumokat, az alkalmazás befejezésekor pedig az egész listát kiürítik, felszabadítva a lefoglalt memóriaterületet (lásd a 4.1. pontban).

Mi történne, ha alkalmazásunkat így írnánk meg?

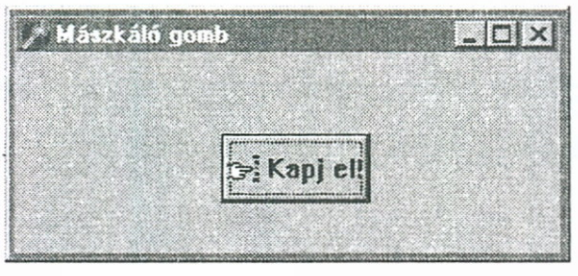
{4_DINAMI\KERDESP.DPR}

```
...
Type
   TForm1 = class(TForm)
        ...
     private
       Cimke: TLabel;
   end;
...

procedure TForm1.FormCreate(Sender: TObject);
begin
  Cimke:= TLabel.Create (Self);
  Cimke.Parent := Self;
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button:
            TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  With Cimke Do
    begin
      Top:= Y;
      Left:= X;
      Caption := '('+IntToStr(X)+', '+IntToStr(Y)+')';
    end;
end;

end.
```

Ebben az esetben végig egy címkével dolgozunk: privát mezőként deklaráljuk az űrlap-osztályban, az űrlap létrehozásakor őt is létrehozzuk, majd kattintáskor a pozícióját átírjuk az egérkurzor pozíciójára.

4.14.4 Mászkáló gomb (*TTimer* használata)

Legyen űrlapunkon egy gomb. A gomb ugráljon össze-vissza az űrlapon belül mindaddig, amíg „el nem kapjuk". Ha kattintottunk rá, akkor álljon meg. Újbóli kattintásra induljon el megint...



4.19. ábra. A gomb addig „mászkál", míg el nem kapjuk

Megoldás (🖫 4_TIMER\PMASZIK.DPR)

Nevezzük el űrlapunkat *frmJatszoter*-nek, cipeljünk rá egy gombot, állítsuk be a feliratát. A gombnak időközönként el kellene mozdulnia valamilyen irányba. De hova írjuk be a gomb mozgását? Erre való a *TTimer* (időzítő). A *TTimer* a komponensnél beállíthatjuk, hogy milyen időközönként (*Interval* jellemző), mi történjen (*OnTimer* esemény). A *System* palettáról fogjunk tehát meg egy *TTimer* komponenst, és helyezzük el az űrlapon. Ez futás közben láthatatlan lesz, így mindegy, hogy hova tesszük. Állítsuk be adatait:

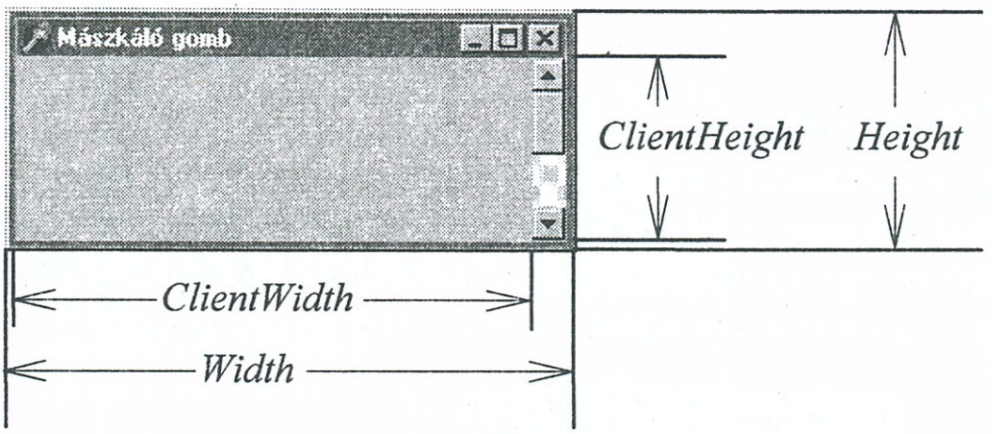
Komponens	Jellemző	Érték
Gomb: TBitBtn	Caption	'Kapj el'
	Glyph	DELPHI\IMAGES\BUTTONS\FIND.BMP
Timer:TTimer	Interval	100 *{ezredmásodpercben}*
	Enabled	True

Az *OnTimer* esemény tizedmásodpercenként következik be. Ebben fogjuk a gombot ±10 pixellel elmozdítani X és Y irányban.

```
procedure TfrmJatszoter.TimerTimer(Sender: Tobject);
Var rx,ry:Integer;
begin
   rx:=10*(1-random(3));  {⇒ rx  = -10, 0 vagy +10 }
   ry:=10*(1-random(3));
   If (gomb.Left+ rx >0) And
      (gomb.Left+ rx+ gomb.Width < ClientWidth) Then
      {ha nem hagyja el az űrlapot bal vagy jobb szélét}
      gomb.Left:= gomb.Left+ rx;
   If (gomb.Top+ ry >0) And
      (gomb.Top+ ry+ gomb.Height < ClientHeight) Then
      {ha nem hagyja el az űrlap felső vagy alsó peremét}
      gomb.Top:= gomb.Top+ ry;
end;
```

A gomb tehát jelenlegi pozíciójából tizedmásodpercenként elmozdulhat 10 pixeles körzetben. Csak akkor mozdítjuk el ténylegesen, ha az új pozíció még az űrlap belsejében van. Ehhez az űrlap *ClientWidth* és *ClientHeight* jellemzőit használjuk, mivel ezek – a *Width* és *Height* jellemzőktől eltérően – a ténylegesen használható terület méreteit tartalmazzák (4.20. ábra).

Az *OnTimer* eseményre nem építhetünk be hosszadalmas tevékenységeket. Egy feldolgozásnak be kell fejeződnie még mielőtt egy újabb időzítőesemény bekövetkezne.



4.20. ábra. Az űrlap méreteinek lekérdezése

A véletlenszám-generátort a program elején inicializálnunk kell (ennek köszönhetően a *Random*-mal generált értékek ténylegesen „véletlenek", különbözőek lesznek):

```
procedure TfrmJatszoter.FormCreate(Sender: Tobject);
begin
  Randomize;
end;
```

Amikor kattintunk a gombra, a következő két eset lehetséges:

- ha eddig mozgott, akkor most meg kellene állnia,
- ha viszont eddig állt, akkor most kellene elindulnia.

Ezt a hatást az időzítő ki-be kapcsolásával érjük el, hiszen ha az időzítőt kikapcsoljuk, akkor az *OnTimer* esemény sem következik többet be, tehát a gomb mozdulatlan marad.

```
procedure TfrmJatszoter.gombClick(Sender: TObject);
begin
  Timer.Enabled:= Not Timer.Enabled;
end;
```

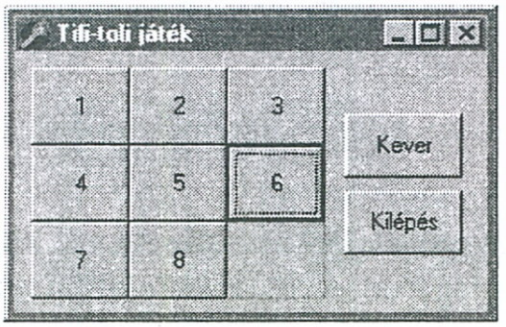
Változtassa meg a kódot annak érdekében, hogy leálláskor a gomb felirata változzon meg „Indíts el!" -re!

Fejlessze tovább a játékot! Lehessen beállítani, hogy milyen gyorsan mozogjon a gomb és hány kattintás után álljon le. Leálláskor írja ki, mennyi idő alatt „fogták meg".

Még egy ötlet: ne csak az elmozdulás iránya, hanem ennek mértéke, sőt esetleg az egyes ugrások közötti időintervallum is legyen véletlenszerű!

4.14.5 Tili-toli játék

Kezdetben a számok rendezetlenek. Ezeket helyre kell tologatni úgy, hogy a számok növekvő sorrendben helyezkedjenek el, és a lyuk az utolsó helyen álljon. A tologatás abból áll, hogy a lyuk helyére egy szomszédos számot helyezünk el. A *Kever* gomb keverjen egyet a számokon úgy, hogy a lyukat is teljesen véletlenszerű pozícióba tegye.



4.21. ábra. Tologassuk helyre a számokat!

Megoldás (🖫 4_TILITO\PTILITO.DPR)

Kezdjük a legelején! Hogyan és főleg miből rakjuk ki a 3x3-as számmátrixot? Hogyan működjön a játék? Ha kattintással akarjuk a számokat mozgatni, akkor máris megvan a megoldás: vegyünk fel 9 gombot, a számok pedig legyenek ezek feliratai.
Van azonban egy másik probléma: a 9 gombot tervezési időben helyezzük el az űrlapon? Ennek a megoldásnak két hátránya is van:

- először is tervezéskor túl munkás lenne a gombokat pont egymás mellé igazítani,
- másodszor pedig, ha picit belegondolunk a keverésbe, akkor azt tapasztaljuk, hogy a gombokat keveréskor ciklusban kell feldolgoznunk: minden gombra új feliratot fogunk generálni. Ha viszont tervezéskor hozzuk ezeket létre, akkor külön neveik lesznek (*Gomb1*, *Gomb2*...), ezzel lehetetlenné téve egységes feldolgozásukat.

Hozzuk tehát létre a gombokat futáskor, valamikor a program legelején (űrlapunk létrehozásakor – *frmTiliToli.OnCreate*). Most már csak azt kellene eldönteni, hogy milyen adatszerkezetben tároljuk a gombokat. Talán a legjobb megoldás egy 3x3-as gombmátrix lenne. Hol deklaráljuk ezt a mátrixot? Elég, ha ezt a létrehozás metódusában tesszük? Természetesen nem, hiszen más metódusoknak is hozzá kell majd férniük a gombokhoz (például keveréskor). Legyen tehát a gombmátrix az űrlaposztály privát mezője. A lyuk mátrixbeli pozíciója keveréskor dől el, így azt is tároljuk a *Lyukx*, *Lyuky* mezőkbe.

```
Type
  TfrmTiliToli = class(TForm)
      Panel: TPanel;
```

```
        btnKever: TButton;
        btnKilepes: TButton;
        procedure FormCreate(Sender: TObject);
        procedure btnKeverClick(Sender: TObject);
        procedure btnKilepesClick(Sender: TObject);
      private
        Gombok: Array[1..3,1..3] of TButton;
        Lyukx, Lyuky: Byte;
        procedure Keveres;
        procedure GombClick(Sender:TObject);
  end;
```

A gombok létrehozásának kódja:

```
procedure TfrmTiliToli.FormCreate(Sender: Tobject);
var i,j:Integer;
begin
  For i:=1 To 3 Do
    For j:=1 To 3 Do
      begin
        Gombok[i,j]:= TButton.Create(self);
        With Gombok[i,j] Do
              begin
                Parent:= Panel;
                Width:= Panel.Width div 3;
                Height:= Panel.Height div 3;
                Left:= (i-1)*Width;
                Top:= (j-1)*Height;
                OnClick:= Gombclick;
              end;
      end;
  Keveres;
end;
```

A gombokat egy tervezési időben űrlapra helyezett *Panel*-re tesszük! Ennek két oka is van:

- A gombok pozícióját ne az űrlap szélétől kelljen megadnunk. Ha a gombok szülője a *Panel*, akkor az [1,1]-es gomb bal felső sarka a (0,0) koordinátákban van. Így rövidebb a gombok pozícióját kiszámoló képlet, és ugyanakkor kódunk rugalmasabbá válik: ha tervezéskor elmozdítjuk, vagy átméretezzük a *Panel*-t, akkor a program az új pozícióban és az új méretben fogja a gombokat megjeleníteni anélkül, hogy a programban bármit is változtatnunk kellene.
- A lyuknak megfelelő gomb láthatatlan lesz; ugyanakkor viszont jó lenne látni a helyét, a gombmátrix határait (lásd 4.21. ábra). A *Panel* erre is jó.

A gombok létrehozása után keverünk is rajtuk egyet. Erre írtuk a *Keveres* metódust. Meghívjuk a *TfrmTiliToli.OnCreate* végén és a *btnKeveres* gomb kattintására.

```
procedure TfrmTiliToli.Keveres;
var SzamHalmaz:Set of Byte;
    Szam, i, j:Byte;
begin
   Randomize;
   SzamHalmaz:= []; {Ide „dobjuk be" a már kiosztott számokat. Ily
                     módon nem lesznek ismétlések}
   For i:= 1 To 3 Do
      For j:=1 To 3 Do
         begin
            Repeat
               Szam:= random(9); {0 jelenti a lyukat; 1..8 jó számok}
            Until Not (Szam in SzamHalmaz);
            SzamHalmaz:= SzamHalmaz + [Szam]; {halmazegyesítési művelet}
            Gombok[i,j].Caption:= IntToStr(Szam);
            If Szam = 0 Then  {ha ez a lyuk ... }
               begin
                  Gombok[i,j].Visible:= False;
                  Lyukx:= i;
                  Lyuky:= j;
               end
            Else
               Gombok[i,j].Visible:= True;
         end;
end;
```

És akkor most jöhet a lényeg! Hogyan fognak a gombok helyet cserélni a lyukkal? Minden gombnak kattintáskor ugyanaz a feladata: „körbenéz", hogy mellette van-e a lyuk, és ha igen, akkor helyet cserél vele. A gombok *OnClick* eseményére az általunk privátként deklarált *GombClick* metódust írjuk be (lásd *TfrmTiliToli.OnCreate*). Íme a metódus kifejtése:

```
procedure TfrmTiliToli.GombClick(Sender:TObject);
   Function LyukSzomszed(Gomb:TButton):Boolean;
      var Lyuk:TButton;
      Begin
         Lyuk:= Gombok[Lyukx, Lyuky];
         LyukSzomszed:=  ((Gomb.Left = Lyuk.Left) And
                          (Abs(Gomb.Top - Lyuk.Top)=Gomb.Height)) or
                         ((Gomb.top = Lyuk.top) And
                          (Abs(Gomb.Left - Lyuk.Left)=Gomb.Width));
      End;
var Seged:Integer;
begin
   if LyukSzomszed(Sender as TButton) then
      begin
         {felcseréljük a lyuk es a gomb pozícióját}
         Seged:= (Sender as TButton).Left;
         (Sender as TButton).Left:= Gombok[Lyukx,Lyuky].Left;
         Gombok[Lyukx,Lyuky].Left:= Seged;
         Seged:= (Sender as TButton).Top;
```

```
        (Sender as TButton).Top:= Gombok[Lyukx,Lyuky].Top;
        Gombok[Lyukx,Lyuky].Top:= Seged;
    end;
end;
```

Miért van szükség a *GombClick* metódusban típusátalakításra (*(Sender as TButton).Left*)? Azért, mert a *Sender* paramétere *TObject* típusú, ebben az osztályban pedig még nincs *Left* és *Top* jellemző. De miért nem deklaráljuk inkább a *Sender*-t *TButton*-nak, hiszen a *Sender* – mint az üzenet okozója – a mi esetünkben mindig egy gomb lesz? Ezt nem tehetjük. Fordítási hibához vezetne a *Gombok[i,j].OnClick:= GombClick* értékadásnál. Hát persze, hiszen a *TButton* osztály *OnClick* eseményjellemzője *TNotifyEvent* típusú, ami nem más, mint:

```
type TNotifyEvent = procedure (Sender: TObject) of object;
```

Így hát a *GombClick*-nek is ilyen típusúnak kell lennie.

Fejlessze tovább a játékot! A számok helyes kirakása esetén gratuláljon a gép! Továbbá, lehessen beállítani a mátrix méretét (3x3-as, 4x4-es ...).

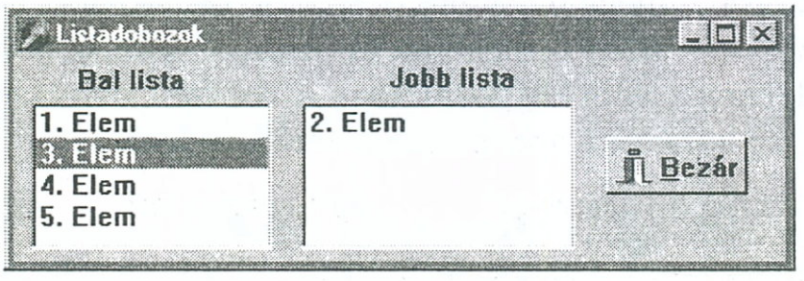
4.15 Drag&Drop (fogd és vidd, vonszolás) technika

Windowsban már megszoktuk, hogy vonszolással bizonyos elemeket át lehet helyezni/másolni más helyekre (Például szövegek mozgatása/másolása Winwordben, vagy állományok másolása/áthelyezése a fájlkezelőben, táblák kapcsolatainak megadása Access-ben ...). Delphiben is van erre lehetőség. Minden vezérlőelem (*TControl* leszármazottja) rendelkezik a következő elemekkel:

- *DragMode* és *DragCursor* jellemzőkkel,
- *BeginDrag* metódussal és
- *OnStartDrag, OnDragOver, OnDragDrop* és *OnEndDrag* eseményjellemzőkkel.

Ezek segítségével Delphiben tulajdonképpen akármit vonszolhatunk: listadobozok sorait, képeket, színeket, állományokat, gombokat...

A vonszolás tanulmányozására oldjuk meg a következő feladatot: a listaelemeket lehessen a jobb listából átvonszolni a balba és fordítva.



4.22. ábra. A listaelemek vonszolása

Megoldás (🖫 4_DRGDRP\LISTAP.DPR)

A vonszolás három fontos lépésben valósul meg:

1. **Vonszolás elkezdése**: a felhasználó lenyomja az egér bal gombját a vonszolandó objektum fölött (a mi esetünkben a listadoboz egy során). A művelet elindítása a vonszolandó objektum (listadoboz) **DragMode** jellemzőjének értékétől függ:
   - *DragMode = dmAutomatic* ⇨ azonnal, automatikusan elkezdődik a vonszolás
   - *DragMode = dmManual* ⇨ a vonszolás csak a kódból meghívott **BeginDrag(Flag:Boolean)** metódus hatására következik be: *BeginDrag(True)* hatására rögtön a metódushívás után, *BeginDrag(False)* hatására pedig csak egy legalább 5 pixeles egérelmozgatás után.

   Általában a kézi vonszolást használjuk (*DragMode = dmManual*), hiszen ez a megoldás lehetőséget nyújt a helyzet elbírálására, esetleg közbeszólásra. Példánkban, csak akkor van értelme a vonszolásnak, ha az egérgomb lenyomásának pozíciójában ténylegesen található elem (nem létező sort nem tudunk elvonszolni). A vonszolás indítását pedig általában a *BeginDrag(False)*-szal váltjuk ki, így egy egyszerű kattintás nem fog vonszolásba torkollani.

   melyik vezérlőelemre kattintottak

```
procedure TfrmVonszolas.ListaMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X,Y: Integer);

                                          hova kattintottak
begin
  {ha a felh. lenyomta az egér bal gombját a listadoboz fölött}
  If (Button = mbLeft) And (Sender is TListBox) Then
     With Sender as TListBox Do
        If ItemAtPos(Point(X,Y), True) >=0 Then
           BeginDrag(False);
end;
```

A vonszolásnak mindkét irányban működnie kell. Ennek érdekében mind a bal, mind a jobb lista *OnMouseDown* eseményjellemzőjét le kell kódolnunk. Ha kódunkban a *Sender* paramétert használjuk, és nem égetjük be a listadobozok neveit, akkor ugyanezt a kódot mindkét listadoboz esetén használhatjuk.

2. **Vonszolás fogadása**: minden objektum, ami fölött vonszolunk, *OnDragOver* üzenetet kap. Az *OnDragOver* üzenetkezelőben az *Accept* változó paramétert igazra vagy hamisra kell állítanunk, attól függően, hogy a vonszolt objektumot fogadjuk-e vagy sem. (Vigyázat, a paraméter alapértelmezett értéke az Igaz!)

   Ami fölött vonszolunk *Amit vonszolunk* *Egér pozíciója*

```
procedure OnDragOver(  Sender, Source: TObject; X, Y: Integer;
                       State: TDragState; var Accept: Boolean);
```

   Vonszolás állapota *Képes-e fogadni, vagy sem*

A vonszolás állapota (*State*) *dsDragEnter, dsDragMove* vagy *dsDragLeave* értékű, attól függően, hogy most éppen beléptünk a célterületre, a célterületen belül vonszolunk, vagy pedig éppen készülünk innen kilépni.

Legtöbbször a fogadóképesség a vonszolt objektum típusától függ. A mi példánkban is, egy listadoboz csak egy **másik listadoboz** elemeit képes fogadni:

```
procedure TfrmVonszolas.ListaDragOver(Sender, Source: TObject;
        X,Y: Integer; State: TDragState; var Accept: Boolean);
begin
   Accept:= (Source is TListBox) And (Source <> Sender);
end;
```

Az *Accept* jellemző igazra állításával az egérkurzor formája is megváltozik: felveszi a vonszolt objektum *DragCursor* jellemzőjében megadott értéket (például *crDrag* ⇨ , *crMultiDrag* ⇨ ...). Ott pedig, ahol semmi sem fogadja, az egérkurzor is „tagadó": *crNoDrop* ⇨ . (Saját rajzolású kurzorformák felhasználását lásd a 4.16. pontban.)

3. **Dobás és befejezés**: ha az egérgomb felengedése egy fogadóképes objektum fölött következik be, akkor:
   - Az objektum, amely fölött a dobás bekövetkezett, kap egy *OnDragDrop* üzenetet.
   - A vonszolt objektum egy *OnEndDrag* üzenettel értesül a fogadtatásáról.

Az *OnDragDrop* eseményjellemzőben szoktuk feldolgozni a vonszolt objektumot, az *OnEndDrag*-be pedig esetleges frissítéseket lehet kezdeményezni (ha például egy állományt áthelyeztünk egy másik könyvtárba, akkor az eredeti könyvtár tartalmának frissülnie kell).

Példánkban az *OnDragDrop*-ban helyezzük át a listaelemet, az *OnEndDrag*-ben pedig egy információs szöveget jelenítünk meg:

A fogadó objektum *A vonszolt objektum*

```
procedure TfrmVonszolas.BalListaDragDrop (Sender, Source:
                  TObject; X, Y: Integer);
```

Egér pozíciója

```
begin
  KijeloltetViszi(Source as TListbox, Sender as TListBox);
end;
```

A *KijeloltetViszi* metódus azonos a 4.14.2. pontban tárgyalttal: ez első paraméterként kapott listadoboz kijelölt elemeit áthelyezi a második paraméterben megadott listadobozba.

A vonszolt objektum *A fogadó objektum*

```
procedure TfrmVonszolas.ListaEndDrag (Sender, Target: TObject;
                                      X, Y: Integer);
```

Egér pozíciója

```
begin
  if Target <> Nil Then {sikeres fogadtatás esetén}
     ShowMessage(Format('A %s objektum fogadta a (%d, %d)'+
     'koordinátákban',[(Target as TComponent).Name, X, Y]));
end;
```

Vonszoláskor mindvégig rendelkezésünkre áll a vonszolást kezdeményező (forrás) komponens (a mi esetünkben ez egy listadoboz volt, de lehetett volna bármi más is). A vonszolás eredménye a célkomponens *OnDragDrop* metódusában dől el: itt áthelyezhetünk egy sort a forrásból, de ugyanúgy átvehetnénk csak a színét vagy betűtípusát... A lehetőségek tehát korlátlanok!

4.16 Egyéni kurzorok. A TScreen osztály

Delphiben minden vezérlőelem (*TControl* leszármazottja) rendelkezik egy ***Cursor*** és egy ***DragCursor*** jellemzővel. Az elsővel beállíthatjuk, hogy milyen legyen az egérkurzor akkor, amikor az egér a vezérlőelem fölött tartózkodik, a második pedig az illető komponens vonszolása közbeni kurzorformáját adja meg. De vajon milyen értékeket vehetnek fel ezek? E kérdés megválaszolásának érdekében ismerkedjünk meg a *Screen* objektummal:

Delphiben a ***Screen*** objektum a képernyő pillanatnyi állapotával kapcsolatos jellemzőket tartalmazza. Ez nem egy komponens, melyet lecipelünk űrlapunkra, vagy amit futáskor kellene dinamikusan létrehoznunk. A *Screen* egy globális objektum (akárcsak az *Application* is), mely automatikusan létrejön az alkalmazás indításakor, jellemzőit pedig az egész programban lekérdezhetjük, állíthatjuk. Tekintsük át ezeket röviden:

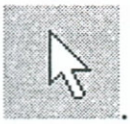
4.16.1 TScreen osztály

- Fontosabb jellemzői:
  - **ActiveForm: TForm**
    Egy alkalmazásnak több űrlapja is lehet. Egy adott pillanatban ezek közül egyesek láthatóak, mások nem. Az összes látható űrlap közül egyszerre csak egy lehet fókuszban, a *Screen.ActiveForm* jellemző ezt tartalmazza.
  - **ActiveControl**: az aktív űrlapon levő aktuális vezérlőelem
  - **Cursors[I:Integer]**: a rendelkezésünkre álló kurzorformák
  - **Cursor: Integer**

Ez egy egész szám, pontosabban egy *Cursors* tömbbeli indexérték, mely az éppen aktuális kurzornak felel meg. Módosításával válthatjuk a kurzort (lásd a következő példában).

- **FormCount**: az aktuális alkalmazás űrlapjainak száma
- **Forms**: az űrlapok listája
- **Height, Width**: a képernyő felbontása (pixelekben)

A *Cursors* tömb a -21→0 indexei között eleve tartalmaz néhány beépített kurzorformát:

Cursors[-1] (crNone)⇨ , *Cursors[-2] (crArrow)*⇨  ...

A *Screen.Cursor* jellemző adja meg az aktuális kurzor indexét. Ha értéke 0 (*crDefault*), akkor hagyja érvényesülni az űrlapnál és a felületén elhelyezett komponenseknél beállított kurzorformákat. Ha viszont más értékkel látjuk el, akkor felülírja a vezérlőelemek egyenkénti beállításait.

Alkalmazásainkba egyéni kurzorformát a következőképpen állíthatunk be:

1. Először elő kell állítanunk magát a kurzort. Ez állhat egy kész *.CUR vagy *.ANI (animált kurzor) állományból, de lehet ez egy *.RES (*resource*) állomány is, amiben megtalálható a kurzorunk. Ha pedig mi szeretnénk saját kurzort rajzolni, akkor ezt a Delphihez tartozó Image Editor segédprogrammal tegyük: hozzunk létre benne egy új RES vagy CUR állományt, és rajzoljuk meg a kívánt kurzorokat. A kurzor RES állományban történő elhelyezésének két előnye is van a többi állománytípushoz képest: egyrészt egy RES állományban több kurzort is elhelyezhetünk, másrészt pedig az elkészített RES állományt később a kurzorokat használó alkalmazás EXE állományába is beépíthetjük (a főprogramba elhelyezett *{$R ÁllományNév.RES}* fordítási direktívával). Ez azt jelenti, hogy az alkalmazás egy másik gépre történő telepítésénél nem kell külön még a kurzorok állományaira is figyelnünk.
2. A következő lépés a kurzor programból történő megjelenítése. Egy kurzort Delphiből két lépésben lehet megjeleníteni:
   - Előbb be kell töltenünk a kurzort a *Screen.Cursors* tömb egy rekeszébe (például *Screen.Cursors[1]*). Lehetőleg ne írjuk felül a létező kurzorokat, ezek negatív indextartományban találhatók. A kurzor betöltése vagy a *LoadCursor*, vagy a *LoadCursorFromFile* API függvényhívással történik, attól függően, hogy a kurzor az alkalmazás EXE, vagy pedig egy külső CUR vagy ANI állományból származik. Például:

```
{Ha a kurzor az EXE-ből származik, akkor a projektállomány elején megtalálható a következő fordítási direktíva}
{Cursors.RES a kurzorainkat tartalmazó állomány}
{$R Cursors.RES}
```

```
{A SAJATKURZOR nevű kurzor betöltése a következőképpen történik}
Cursors[1]:= LoadCursor(hInstance, 'SAJATKURZOR');

{Ha a kurzor egy külső állományból származik, akkor így töltsük be:}
Cursors[1]:= LoadCursorFromFile('SAJATKURZOR.CUR');
```

- Ha csak egy adott vezérlőelem felett akarjuk megjeleníteni az új kurzort, akkor a vezérlőelem *Cursor* jellemzőjét írjuk át arra a tömbbeli indexre, ahova kurzorunkat elhelyeztük. Ha globálisan, az egész űrlapra akarjuk érvényesíteni az új kurzorformát, akkor a *Screen.Cursor* jellemzőjébe írjuk ugyanezt. Ekkor viszont a vezérlőelemek egyenkénti kurzorbeállításai elvésznek.

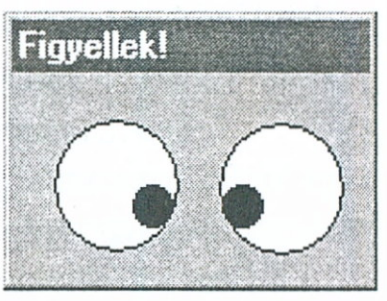
```
Screen.Cursor := 1;
```

Tekintse át a 4_SCREEN\PSCREEN.DPR alkalmazást. Ebben egyrészt látható a *Screen* objektum jellemzőinek használata, másrészt pedig kipróbálható több saját rajzolású és állományból betöltött kész kurzor is.

Feladatok

Egeret követő szempár (🖫 4_SZEMEK\SZEMEKP.DPR)
Készítsen egy alkalmazást, melynek űrlapján egy szempár állandóan kövesse az egérkurzor mozgását. Az űrlapot ne lehessen eltüntetni, ez mindig a többi ablak felett helyezkedjen el (*FromStyle = fsStayOnTop*).

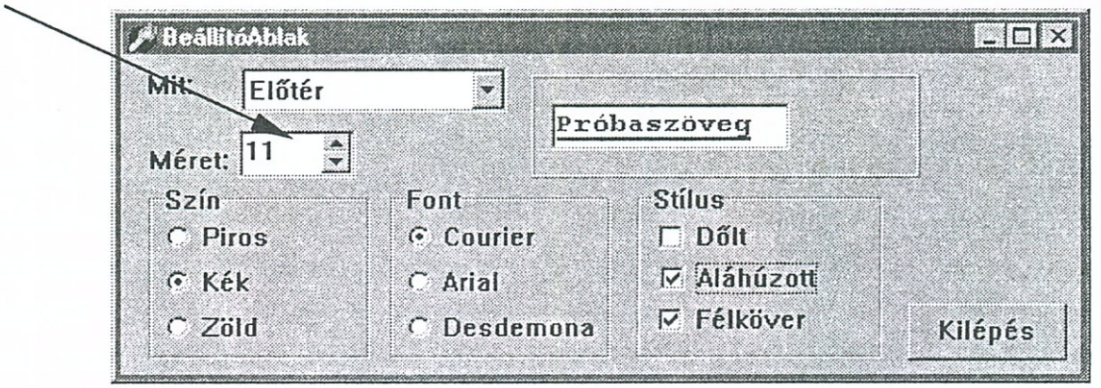
Tipp: A szemeket 2-2 *TShape* (*Additional paletta*) komponensből rakja ki. Állítsa be a formájukat (*Shape*) és színüket (*Brush.Color*). Az egér mozgására csak a belső két *TShape* komponens pozícióját változtassa.



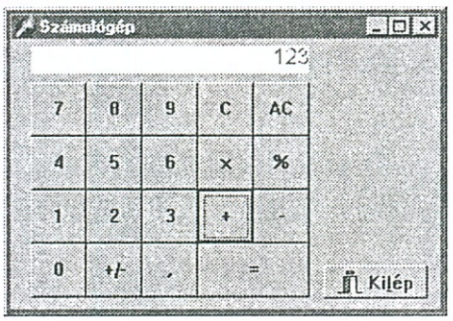
Figyelem, a megoldáshoz mértani ismereteinkre is szükségünk van!

Beállítóablak (🖫 4_BEALL\BEALLPR.DPR)
Készítse el a következő beállítóablakot. A beállításokat a próbaszövegen azonnal lehessen követni. A *Mit* kombinált listában kiválasztjuk a beállítandót: előtér vagy háttér. Előtérnek számít a betű színe, típusa, stílusa, háttérnek pedig a szerkesztődoboz háttérszíne. Háttérállítás esetén csak a *Szín* választógomb csoport legyen elérhető. Minden egyéb szürküljön be.

TSpinEdit (*Samples* paletta)



Számológép
Írjon egy számológép programot!



Tipp: A gombokat futás közben helyezze el egy *Panel*-re. Csoportosítsa az azonos szerepű gombokat!

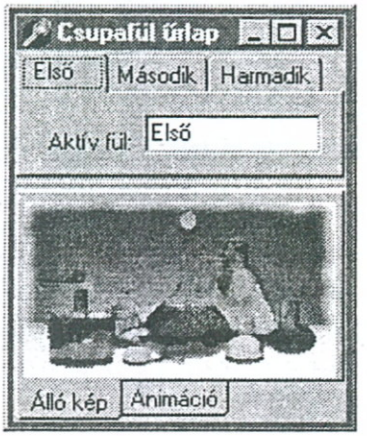
Aknakereső (🖫 4_MINESWEEPER\PMINE.DPR)
Készítse el a Windows rendszerekben megtalálható aknakereső (*minesweeper*) játékot!
Tipp: Celláit vagy *TSpeedButton*, vagy *TPanel* komponensekből rakja ki futási időben.
Vigyázat, a felderítő rekurzív metódus könnyen veremhibához vezethet[1].

[1] A 16 bites Delphi-ben a verem és adatszegmens együttes mérete nem haladhatja meg a 64 KByte-ot. A 32 bites Delphi-ben a verem mérete gyakorlatilag nincs korlátozva (max. 2GBájtos lehet).

5. Több info az alkalmazásban

Ha alkalmazásunk már egy kicsit nagyobb, több információt kell benne megjelenítenünk, és ha ez már nem fér el egy űrlapon, akkor két irányba indulhatunk el:

- A logikailag összetartozó információkat egy űrlapon több oldalra helyezhetjük el. Mindig csak egy oldal látható, egy másikat a fülére való kattintással lehet megjeleníteni (lásd 5.1. ábra).
- Több űrlapos alkalmazást készítünk. A program indításakor megjelenik a főűrlap, róla lehet majd a többi ablakra eljutni.



5.1. ábra.

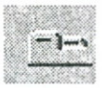
Ebben a fejezetben az 5.1. pontban megismerkedünk a többoldalas űrlapok készítésének lehetőségeivel, az 5.2. pontban pedig többablakos alkalmazásokat fogunk létrehozni.

5.1 Fülek az űrlapon

A füleket megvalósító komponensek nagyon különböznek a Delphi három verziójában.

- **Delphi 1:** *TTabbedNotebook*, *TNotebook*, *TTabset*
- **Delphi 2:** *TPageControl*, *TTabControl* (de csak „felső füleik" lehetnek). Ha alsó füleket szeretnénk, akkor a Delphi 1-es komponensét kell használni (*TTabset* a Win 3.1 palettáról).
- **Delphi 3:** *TPageControl*, *TTabControl* („felső" és „alsó füleik" is lehetnek). A Delphi 1-es komponenseket már csak nagyon ritkán használjuk, például ha képeket akarunk a fülsoron megjeleníteni (lásd 6. fejezet). A Delphi 1-es komponensei itt is a Win3.1 palettán találhatók.

Ahogyan ez elvárható, az újabb verzió komponensei többet tudnak kevesebb erőráfordítással; ha lehet, akkor ezeket használjuk.

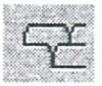
5.1.1 TTabControl (Win32 paletta) 

- Helye az osztályhierarchiában: *TComponent/TControl/TWinControl/...TTabControl*
- Szerepe: fülsor egy lappal. Akkor használjuk, amikor a különböző fülek esetén látható információ szerkezete azonos, csak a tartalma változik. Ilyenkor elég egyszer megtervezni a lap szerkezetét, a fülekre történő kattintáskor pedig váltogatjuk a tartalmat.
- Fontosabb jellemzői:
  - **Tabs:TStrings**: a fülek feliratát tartalmazza külön sorokban. Az itt specifikált sorok száma határozza meg a komponens füleinek számát.
  - **TabIndex**: az aktuális fül indexe (az első fül indexe = 0)
  - **TabPosition**: *tpTop, tpBottom.* a fülek pozíciója
- Fontosabb eseményjellemzői:
  - **OnChange**: fülváltáskor következik be. A *TabIndex* ilyenkor már az új fül indexét tartalmazza.
  - **OnChanging**: fülváltás közben következik be. Az *AllowChange* címszerinti paraméter *False*-ra állításával még meggátolható a váltás. Ebben a metódusban a *TabIndex* jellemző még a régi fül indexét tartalmazza.

5.1.2 TPageControl (Win32 paletta)

- Helye az osztályhierarchiában: *TComponent/TControl/TWinControl/...TPageControl*
- Szerepe: fülsor több lappal. Minden fülhöz tartozik egy lap. Akkor használjuk, amikor a különböző fülekre megjelenítendő információ teljesen más szerkezetű.
- Fontosabb jellemzői:
  - **ActivePage**: az aktuális lap
    Új lapot a helyi gyorsmenü segítségével lehet létrehozni: egér jobb gomb, majd *New Page*. Hatására azonnal megjelenik egy új fül és a hozzátartozó lap. A *Caption* jellemzőbe be kell írnunk az új fül feliratát.
    A lapok közötti váltás akár az egérrel, akár az *ActivePage* jellemző állításával történhet, akár tervezési, akár futási időben.
  - **TabPosition**: azonos a *TTabControl*-nál ismertetettel
- Fontosabb eseményjellemzői:
  - **OnChanging, OnChange:** azonosak a *TTabControl* eseményjellemzőivel

Nézzük most a Delphi 1-beli komponenseket (Delphi 1-ben az *Additional,* míg a Delphi 2-ben és 3-ban a Win3.1 palettán találhatók):

5.1.3 TTabbedNotebook (Win 3.1 paletta) 

- Helye az osztályhierarchiában: *TComponent/TControl/TWinControl/...TTabbedNotebook*
- Szerepe: több oldalas megjelenítés; a „felső fülek" („alsó füleket" nem is tud) segítségével váltunk. (a későbbi *TPageControl* megfelelője).
- Fontos jellemzői:
  - **Pages**: fülek feliratai
  - **ActivePage**: aktív fül felirata
  - **PageIndex**: aktív fül indexe
- Fontos eseményei:
  - **OnChange**: akár kattintással, akár programból kezdeményezzük a fülváltást, az *OnChange* esemény még váltás közben bekövetkezik. Ebben az *AllowChange* paraméter segítségével még meggátolhatjuk a váltást. A *NewTab* paramétere tartalmazza a potenciális új fül indexét.
  - **OnClick**: kattintáskor hívódik meg. Ha a kattintás fülváltást eredményez (például nem az eddig is aktív fülre kattintottunk), akkor ilyenkor a *PageIndex* már az új fül indexét tartalmazza. Figyelem, ha az oldalváltás programból történik, akkor az *OnClick* eseményjellemzőbe írt utasítások nem futnak le!

5.1.4 TNoteBook (Win 3.1 paletta)

- Helye az osztályhierarchiában: *TComponent/TControl/TWinControl/...TNotebook*
- Szerepe: többoldalas füzet. Alapértelmezés szerint nem tartozik hozzá fülsor. Általában együtt használjuk a *TTabset* komponenssel, ez biztosítja számára egy „alsó fülsort".
- Jellemzői:
  - **Pages**: a füzet oldalainak címei
  - **ActivePage**: aktuális oldal címe
  - **PageIndex**: aktuális oldal indexe
- Eseménye:
  - **OnPageChanged**: oldalváltás után következik be

5.1.5 TTabset (Win 3.1 paletta)

- Helye az osztályhierarchiában: *TComponent/TControl/TWinControl/...TTabSet*
- Szerepe: alsó fülsor. Alapértelmezésben egy oldal sem tartozik hozzá. Ha a megjelenítendő információ azonos szerkezetű minden fülre, akkor önmagában használjuk.

Ellenkező esetben összeköthetjük egy *TNotebook* komponenssel (összekapcsolásukat lásd a következő pontban).

- Jellemzői:
  - **Tabs**: füleinek feliratai
  - **TabIndex**: aktuális fül indexe
  - **Style: tsStandard, tsOwnerDraw**
    Ha a *Style* jellemző értéke *tsStandard* (ez az alapértelmezett), akkor a füleken csak szöveg jelenhet meg, ráadásul egyforma stílusban minden fülön. Ha viszont e jellemző értékét *tsOwnerDraw*-ra állítjuk, akkor a füleken megjeleníthetünk különböző stílusú szövegeket és grafikát is (lásd 6. fejezet).
- Eseményei:
  - **OnClick, OnChange**: azonosak a *TTabbedNotebook* eseményeivel

5.1.6 A TNotebook és TTabset együttes használata

Mint már említettük, a *TNotebook* és *TTabSet* komponenseket nagyon gyakran használjuk együtt: a *TNotebook* biztosítja a lapokat, míg a *TTabSet* az alsó füleket. Minden egyes füzetlapnak megfeleltetünk egy azonos nevű fület a fülsoron. Nézzük tehát, milyen jellemzőket kell összeegyeztetnünk, és milyen lépésekben tesszük mindezt:

- Jellemzők megfeleltetése:

TNotebook	TTabset
Pages: TStrings	Tabs: TStrings
PageIndex: Integer	TabIndex: Integer

Teljesen fölösleges kétszer begépelni (tervezéskor) a lapok elnevezéseit: egyszer a füzetnél, másodszor pedig a fülsornál. Elég megtervezni például a füzet oldalait, majd a program elején átírni mindezt a fülsorhoz is (*Fulsor.Tabs:= Fuzet.Pages*).

Futási időben a lapváltás a fülsortól indul, hiszen a fülekre kattintunk. Minden egyes kattintáskor tehát a füzetben is váltanunk kell (*Fuzet.PageIndex := Fulsor.TabIndex*).

- Lépések:
  - Megtervezzük a füzetet, elnevezzük oldalait.
  - Létrehozzuk az alsó fülsort, de nem töltjük ki „füleit".
  - Az űrlap létrehozásakor (*OnCreate*) feltöltjük a fülsor füleit:

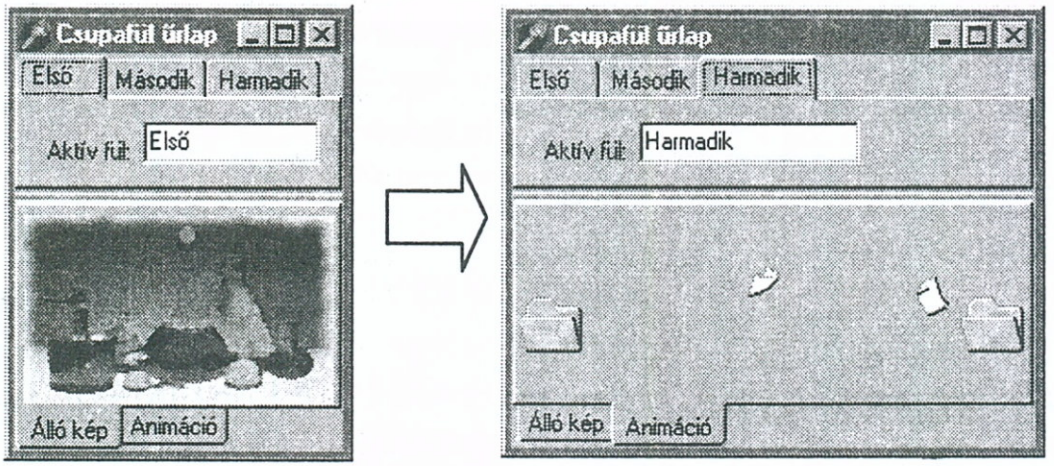
```
Fulsor.Tabs := Fuzet.Pages;
```

  - A fülsorra való kattintáskor pedig a füzetben is lapozunk:

```
Fuzet.PageIndex := Fulsor.TabIndex;
```

5.1.7 Feladat: „Csupafül" űrlap

Gyakorlásképpen készítsük el a következő alkalmazást:



5.2. ábra. „Csupafül" űrlap

Űrlapunk felső és alsó része működésileg egymástól teljesen független. A felső füleket váltogatva az *„Aktív fül"* szerkesztődobozában mindig az aktuális felső fül címét jelenítsük meg. Az alsó fülekre kattintva váltakozva egy álló képet és egy animációt lehessen látni. A felső és alsó részt lehessen egérrel átméretezni.

Megoldás (🖫 5_FULEK\PFULEK.DPR)

Csak a feladat Delphi 3-ban írt megoldását tárgyaljuk részletesen. Delphi 1-re változik némileg a feladatspecifikáció is: elhagyható az átméretezés és az animáció; animáció helyett jelenítsenek meg egy másik képet.

Elemezzük a feladatot!

A felső fülek esetén nincs szükség három különböző lapra. Mindhárom fülnél ugyanazt a szerkesztődobozt láthatjuk, de más-más tartalommal. Erre tehát *TTabControl* komponenst használunk.

Az alsó füleknél más a helyzet. Az első fülnél az álló képet egy *TImage* komponens segítségével jelenítjük meg, míg a másodiknál az animáció megvalósítására *TAnimate* (*Win32* paletta) komponenst kell használnunk. Két, különböző szerkezetű lapra van tehát szükségünk ⇨ a megoldás erre a *TPageControl*.

A két rész futás idejű átméretezését egy ***TSplitter*** (*Additional* paletta) komponenssel oldhatjuk meg. A *TSplitter* használatánál nagyon fontos az igazítás (*Align*).

Tervezzük meg űrlapunkat a következő táblázat szerint:

Komponens	Jellemző	Érték
FelsoFulek: TTabControl	Tabs	Első
		Második
		Harmadik
	TabPosition	tpTop
	Align	alTop
AktivFul:TEdit	Text	''
Splitter: TSplitter	Align	alTop
	Cursor	crVSplit *{függőleges nyíl}*
AlsoFulek: TPageControl	Align	alClient
1. lapjára: AlloKep:TImage	Picture	...\DELPHI\IMAGES\SPLASH\
		256COLOR\CHEMICAL.BMP
	Align	alClient
2. lapjára: Animacio:TAnimate	CommonAVI	aviCopyFiles
	Align	alClient
	Active	True

Most nagyon keveset kell programoznunk:

```
procedure TfrmCsupaful.FelsoFulekChange(Sender: TObject);
begin
   AktivFul.Text:= FelsoFulek.Tabs[FelsoFulek.Tabindex];
end;
```

5.2 Több űrlapos alkalmazások

5.2.1 Alkalmazásunk űrlapjai

A többűrlapos alkalmazások készítésének tanulmányozása előtt foglaljuk össze, milyen ablakokat tartalmazhat egy applikáció:

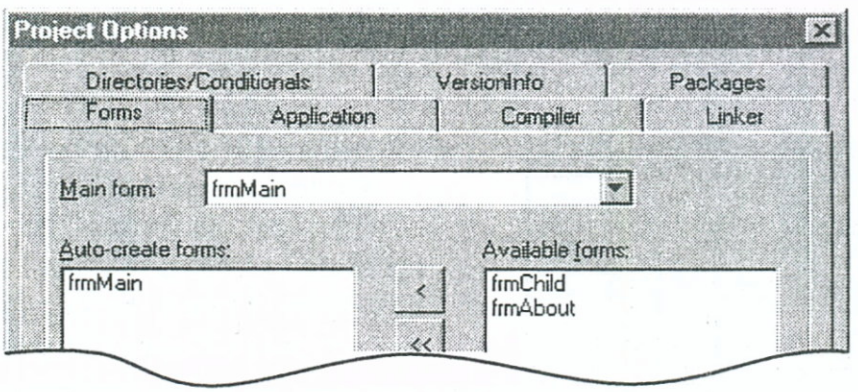
- Egyszerű üzenet és adatbeviteli ablakokat: egy rövid üzenet megjelenítésére, vagy egy szöveg bekérésére nem kell űrlapokat terveznünk, erre a célra használhatunk bizonyos beépített eljárásokat és függvényeket (lásd 5.2.3. pont).
- Windows rendszer által felkínált párbeszédablakokat (*OpenDialog, PrintDialog...*): Delphiből ezeket komponensek segítségével kezelhetjük (lásd 5.2.4. pont).
- Előzőleg megtervezett és mintaként elmentett űrlapokat: bármely űrlapunkat elhelyezhetjük az űrlapminták közé a gyorsmenü *Add to Repository* parancsával. Ha később az elmentett minta alapján szeretnénk egy másik űrlapot létrehozni, akkor a *File/New...* menüpont segítségével, a *Forms* lapon válasszuk ki a mintát (bővebben lásd a 5.2.7. pontban).

- Saját tervezésű űrlapokat: alkalmazásunk sajátos kinézetű űrlapjait legtöbbször helyben kell megterveznünk. Eddigi feladatainkban is ezt tettük. Új űrlapot a *File/New Form* menüpontjával lehet létrehozni.

Űrlapjaink közül

- egyesek automatikusan jönnek létre (***auto-create***), azaz a program indításának pillanatában megszületnek, és ettől kezdve megszűnésükig ott „lapulnak" a memóriában, akkor is, ha pillanatnyilag nem láthatók a képernyőn. Ebben az esetben a létrehozó és megszüntető kódrészletet a rendszer építi be az alkalmazásba, nekünk nincs is vele dolgunk. Újonnan megtervezett űrlapoknál ez az alapértelmezett állapot.
- más űrlapok nem jönnek automatikusan létre, azaz csak megjelenítésük pillanatától a bezárásukig foglalják a memóriát. Ezeket az űrlapokat nekünk kell kódból létrehoznunk, megjelenítenünk, majd bezárásuk után megszüntetnünk (az *auto-create* űrlapoknál csak a megjelenítéssel kellett foglalkoznunk). A ritkán használt űrlapokat (például a névjegyablakot) ajánlatos *nem auto-create*-nek tervezni.

Az *Project/Options*[1] párbeszédablak *Forms* fülén állíthatjuk be azt, hogy az alkalmazás összes űrlapja közül melyik legyen a főűrlap, melyek jöjjenek létre automatikusan és melyek nem (lásd 5.3. ábra).



5.3. ábra. Az űrlapok típusának beállítása

5.2.2 Az ablakok megjelenítési formái

Bármilyen Windows űrlapot két módon jeleníthetünk meg:

- **Modálisan (Modal):** ez a párbeszédablakok tipikus megjelenítési formája. Az űrlap megjelenítésének pillanatától egészen bezárásáig „uralja" az egész alkalmazást, a felhasználó nem használhatja a többi ablakot. Erre egy rövid sípszó is figyelmezteti, valahányszor félrekattint. Ilyen az állomány-megnyitási vagy mentési párbeszédablak is.

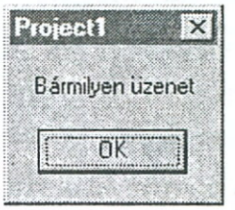
[1] Delphi 1-ben ez a funkció az *Options/Project* menüpontból érhető el.

- **Nem modálisan (Modeless)**: az így megjelenített űrlapról nyugodtan átkattinthatunk alkalmazásunk bármely másik ablakába. A Delphi keretrendszerben ilyen például az objektum-felügyelő ablak, a kódszerkesztő ablak...

5.2.3 Egyszerű üzenet- és adatbeviteli ablakok

- **Üzenetablakok**
  - A **ShowMessage** eljárás egy passzív szöveget jelenít meg. A szöveg egy – csak *Ok* gombot tartalmazó – ablakban bukkan elő. Az ablak címsorában az alkalmazás neve látható.
    Például:

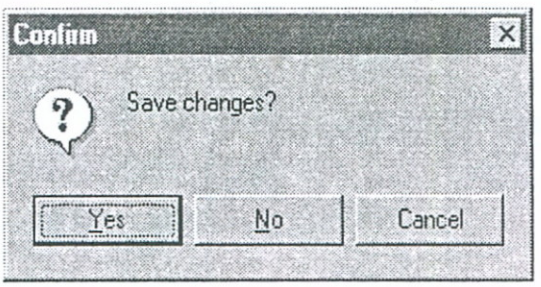
```
ShowMessage('Bármilyen üzenet');
```



  - A **MessageDlg** függvény segítségével egy olyan kérdést tehetünk fel a felhasználónak, melyre egy Igen/Nem, Ok/Mégsem jellegű választ várunk. A függvény deklarációja:

```
Function MessageDlg(Szöveg:String; Ablaktípus: TMsgDlgType;
Gombok: TMsgButtons; SúgóIndex:Integer)
```

Például, egy szövegszerkesztőben a dokumentumablak bezárása előtt ajánlatos rákérdezni a mentésre:

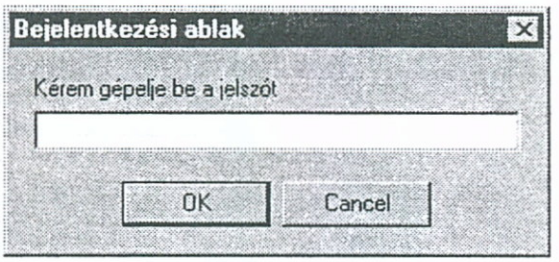



```
Case MessageDlg('Save changes?',
     mtConfirmation,
     [mbYes,mbNo,mbCancel],0) Of
   mrYes:  {Mentés, majd bezárás}
   mrNo:   {Csak bezárás}
   mrCancel:{Vissza az egész,
            se mentés, se bezárás}
End;
```

- **Adatbeviteli ablakok**
  - Az **InputBox** függvény egy karakteres adat bekérését teszi lehetővé. Például egy program futását a következőképpen tehetjük jelszófüggővé:

```
{🖫 5_JELSZO\PJELSZO.DPR}
program pjelszo;
uses Forms,Dialogs, umain in 'umain.pas'{frmJelszavasdi};
{$R *.RES}
begin
   If Inputbox('Bejelentkezési ablak',
     'Kérem gépelje be a jelszót','') = 'A jelszó' Then
     Begin
       Application.Initialize;
```

```
    Application.CreateForm(TfrmJelszavasdi, frmJelszavasdi);
    Application.Run;
  End
 Else
  ShowMessage('Irgum-burgum!');
end.
```



5.2.4 Rendszer-párbeszédablakok használata

Biztosan tapasztalt már a kedves Olvasó olyan „furcsaságot", hogy például magyar Wordben angolul „beszél" az állomány-megnyitási párbeszédablak. Ezzel a jelenséggel akkor találkozhatunk, ha a Windows angol nyelvű, a Word pedig magyar. Magyarázata az, hogy Windowsban a gyakoribb párbeszédablakok közösek, minden alkalmazás meg tudja ezeket jeleníteni: egyesek API függvényhívásokkal, mások (mint például a Delphi alkalmazások) komponensek segítségével. **Azokat a párbeszédablakokat, amelyeket a rendszer kínál fel, és bármely alkalmazás használhat, rendszer-párbeszédablakoknak nevezzük.**

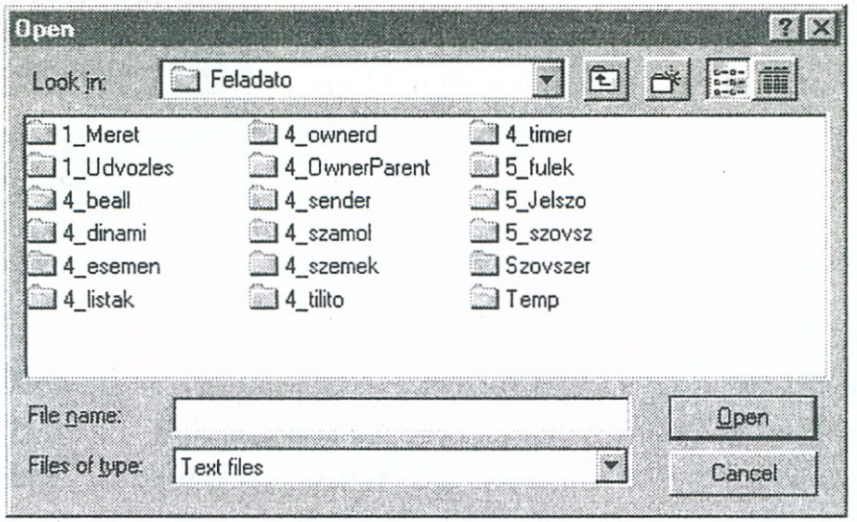
```
TComponent
└─TCommonDialog
   ├─TColorDialog          {színbeállítás}
   ├─TFindDialog           {keresés}
   │  └─TReplaceDialog     {helyettesítés}
   ├─TFontDialog           {betűformázás}
   ├─TOpenDialog           {állomány megnyitás}
   │  └─TSaveDialog        {mentés}
   ├─TPrinterSetupDialog   {nyomtató beállítás}
   └─TPrintDialog          {nyomtatás}
```

5.4. ábra. Párbeszédablak komponensek Delphiben (*Dialogs* paletta)

Delphiben minden rendszer-párbeszédablak számára bevezettek egy-egy komponenst. Ezek mind láthatatlan komponensek, és a *Dialogs* palettán találhatók. Használatuk a következő:

- Helyezzük el a megfelelő párbeszédablak-komponenst arra az űrlapra, amelyikről majd ezt meg szeretnénk jeleníteni.
- Az objektum-felügyelőben állítsuk be adatait (ezek párbeszédablak-specifikusak).
- A megfelelő pillanatban jelenítsük meg a párbeszédablakot az *Execute* metódusa segítségével.

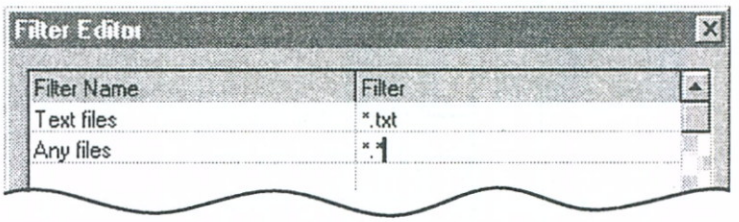
Példának tekintsük át az **állomány-megnyitási párbeszédablak** használatát:



5.5. ábra. Állomány-megnyitási párbeszédablak

Tennivalók:

- Helyezzünk el űrlapunkon egy *TOpenDialog* komponenst; nevezzük el *OpenDialog*-nak.
- Állítsuk be *Filter* jellemzőjét a megfelelő állományszűrőkre (5.6. ábra).
- Állítsuk be *DefaultExt* jellemzőjét is (például TXT-re). Ez lesz az alapértelmezett kiterjesztés.
- Jelenítsük meg a párbeszédablakot:



5.6. ábra. Az állományszűrők beállítása

```
If OpenDialog.Execute Then
  {megjeleníti az ablakot, és ha OK-val léptek ki}
  ShowMessage('A kiválasztott állomány: ' + OpenDialog.Filename);
```

5.2.5 Az alkalmazások típusai

Windowsban alapvetően két típusú alkalmazással találkozhatunk:

- **MDI** (***Multiple Document Interface***) alkalmazás:

  Manapság ilyen elven működik minden dokumentumalapú alkalmazás: Word, Excel, Program manager, File manager...

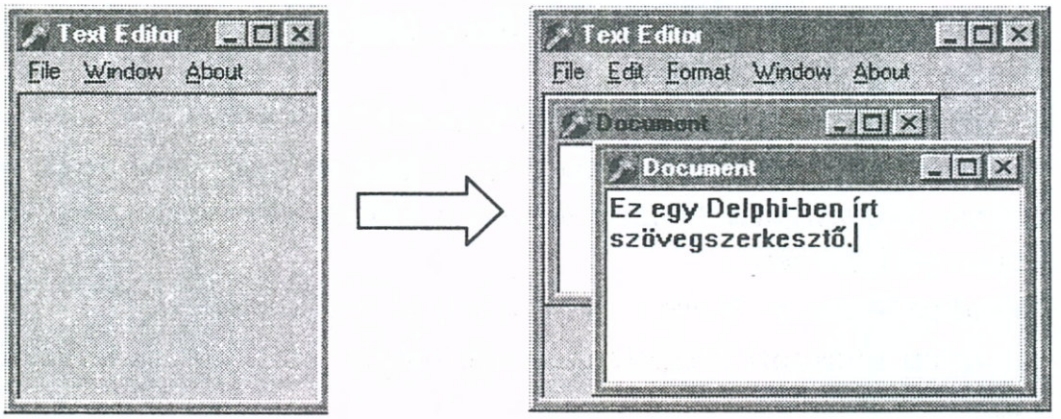
  Egy alkalmazás MDI típusú, ha felügyelete alatt több ablakban jeleníthetünk meg adatokat, grafikát... és azokkal párhuzamosan dolgozhatunk.

  Az MDI alkalmazás részei:

  ↳ Egy főablak (keretablak, *MDIFrame*)

Van címsora, rendszermenüje, főmenüje, eszközsora, állapotsora.

- Akárhány gyerekablak (*MDIChild*). Ezekben jelenítjük meg az adatokat. Van címsoruk, rendszermenüjük, esetleg saját menüjük is. Ebben az esetben a gyerek- és a keretablak menüje összefésülődik.



5.7. ábra. MDI alkalmazás

Szabályok az MDI alkalmazásokban:

- Mindig csak egy gyerekablak lehet aktív.
- A gyerekablakok a keretablak munkaterületén belül helyezkednek el.
- A keretablak bezárása maga után vonja a gyerekablakok bezárását is.

- **SDI** (*Single Document Interface*) alkalmazás:

SDI-nek nevezünk minden egyéb (nem MDI) alkalmazást. Ilyen például a Delphi környezet: van egy főablaka (menüvel, eszköz- és komponenspalettával), de tartalmazza még az objektum-felügyelő, a kódszerkesztő és az űrlaptervező ablakokat is. Ezek mind külön-külön ablakban láthatók, nincs közöttük egyértelmű szülő-gyerek kapcsolat.

5.2.6 A TForm komponens

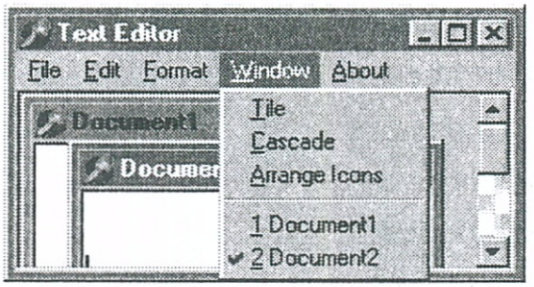
- Helye az osztályhierarchiában:
  TComponent/TControl/TWinControl/TScollingWinControl/...TForm
- Szerepe: az űrlaposztály egy speciális komponens, hiszen magának, a felhasználói interfésznek a megtervezését teszi lehetővé. Azt is tudjuk már, hogy az űrlap tartalmazza az összes ráhelyezett vezérlőelem-objektumot.
- Fontosabb jellemzői:
  - **Caption**: címsorának szövege
  - **BorderIcons:** *[biSystemMenu, biMinimize, biMaximize, biHelp]*
    Itt állíthatjuk be azt, hogy az űrlapnak legyen-e rendszermenüje, minimizáló, maximizáló és súgó gombja?

- **BorderStyle**: bsSizeable, bsDialog...
  A *BorderStyle* jellemzőben az űrlap keretét állíthatjuk be: *bsSizeable* ⇨ méretezhető, *bsDialog* ⇨ párbeszédablakszerű, nem méretezhető keret... Egy képernyővédő űrlapjának esetén ezt a jellemzőt *bsNone* értékre kell beállítani, és ugyanakkor a *BorderIcons* := [] (üres halmaz).
- **ClientWidth**, **ClientHeight**: a kliens (ténylegesen használható) terület méretei (lásd 4. fejezet 4.20. ábra)
- **Position**: *poDesigned, poScreenCenter...*
  A *Position* jellemző értéke az űrlap futáskori megjelenésének pozícióját szabályozza: *poDesigned* ⇨ ahol tervezési időben elhelyeztük, *poScreenCenter* ⇨ a képernyő közepén...
- **WindowState:** *wsNormal, wsMinimized, wsMaximized*
  Az űrlap futáskori megjelenítésének módja: amekkorára terveztük, minimizálva vagy maximizálva.
- **ModalResult**: *mrOk, mrCancel...*
  Egy űrlap bezárása után a *ModalResult* jellemzővel kérdezhetjük le a bezárás okát, azaz, hogy az *Ok*, a *Cancel* vagy más gomb miatt tünt-e el. E jellemző értéke az űrlap bezárását okozó gomb *ModalResult* jellemzőjéből származik (lásd 4. fejezet: *TButton.ModalResult*).
- **FormStyle**: *fsNormal, fsMDIForm, fsMDIChild, fsStayOnTop*
  Tartalmazza az űrlap stílusát. Eddig legtöbbször az *fsNormal*-t használtunk, csak az „egeret követő szempár" feladatnál állítottuk át *fsStayOnTop*-ra. Hatására űrlapunk mindig a többi ablak „tetején" maradt. Az MDI alkalmazások keretablakának kötelezően *fsMDIForm* stílusúnak, míg a gyerekablaknak *fsMDIChild* stílusúnak kell lennie.

- A további jellemzők csak MDI keretablak esetén érvényesek[1]:
  - **MDIChildCount**: tartalmazza a pillanatnyilag megnyitott gyerekablakok számát
  - **MDIChildren[I:Integer]:TForm** jellemző, mely biztosítja a gyerekablakok elérését: *MDIChildren[0.. MDIChildCount-1]*
  - **ActiveMDIChild:TForm** az aktív gyerekablak
  - **WindowMenu:TMenuItem**
    Az MDI alkalmazásokban általában a *Window* menüpont alján meg szokott jelenni a pillanatnyilag nyitott gyerekablakok listája (lásd 5.8. ábra). A Delphiben fejlesztett applikációkban mindez be van építve a rendszerbe. Egyszerűen csak annyit kell tennünk, hogy a keretablak *WindowMenu* jellemzőjébe beállítjuk a majdani gyerekablakok listáját tartalmazó menüpont nevét.

---

[1] Az említett jellemzők a nem MDI keretablakoknál is léteznek, de ilyenkor ezeket természetesen nem alkalmazzuk.



5.8. ábra. A Window menüpont alatt látható a nyitott gyerekablakok listája

- Fontosabb metódusai:
  - **Show**: eljárás, mely megjeleníti az űrlapot
  - **ShowModal**: függvény, mely megjeleníti modálisan az űrlapot, majd vár, amíg ezt bezárjuk. Ekkor visszatérési értéke a bezárás okát tükrözi. Például:

```
If ParbeszedAblak.Showmodal = mrOK Then
   ShowMessage('Az OK gombra kattintottak')
Else
   ShowMessage('A Cancel gombra kattintottak');
```

Van még egy lényeges különbség a *Show* és *ShowModal* között. Vizsgáljuk át az 5.9. ábrán látható programrészleteket! A *Show* metódus megjeleníti az űrlapot, és **rögtön utána** végrehajtja az U1 utasítást. A *ShowModal* is megjeleníti az űrlapot, de utána vár ennek bezárásáig; így az U2 utasítás csak az ablak eltűnése után fog lefutni.
Tehát: a *Show* nem használható akkor, amikor a megjelenítést követő utasításokban (*U1, U2*) hivatkozni szeretnénk az űrlapon beállítottakra. Ilyenkor használjuk a *ShowModal*-t.

`frmAkarmi.Show;` `U1;`	`frmAkarmi.ShowModal;` `U2;`

5.9. ábra. A *Show* és *ShowModal* metódusok összehasonlítása

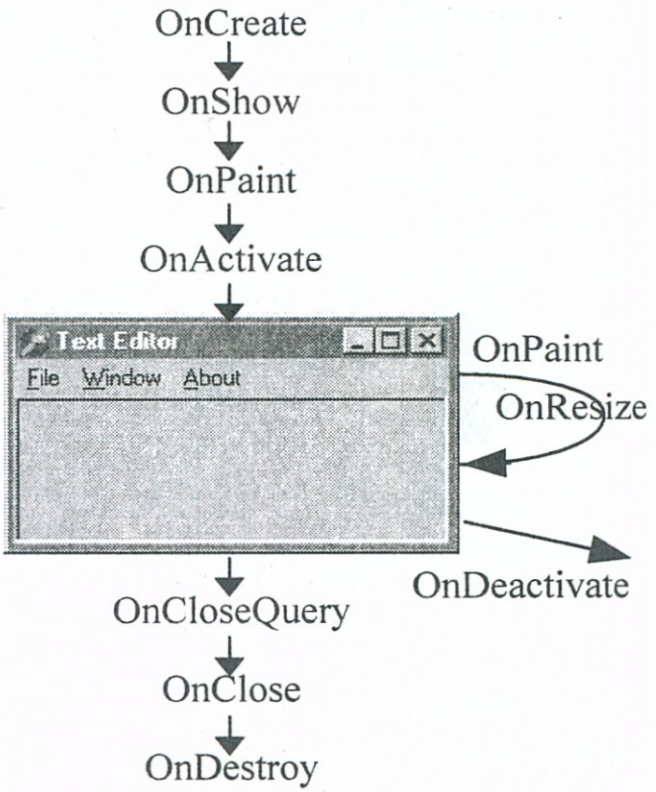
  - **Print**: kinyomtatja az űrlapot
  - **Close**: bezárja az űrlapot. Ha egy alkalmazás főablakát zárjuk így be, akkor az alkalmazás is befejeződik.
- MDI keretablak esetén érvényes metódusok:
  - **Next**: aktivizálja a következő gyerekablakot
  - **Tile**: mozaikszerűen rendezi a nyitott gyerekablakokat
  - **Cascade**: lépcsőzetesen rendezi el a nyitott gyerekablakokat

- **ArrangeIcons**: az ikonméretre minimizált gyerekablakokat rendezi

- Az űrlapok fontosabb eseményei:
  - **OnCreate**: inicializáláskor
  - **OnShow**: megjelenítéskor
  - **OnActivate**: aktivizáláskor
  - **OnPaint**: újrarajzoláskor

Figyelem! Ha azt szeretnénk, hogy egy ábra mindig az űrlapon legyen, átméretezés után se tűnjön el, akkor az ábra kirajzolásának kódját az űrlap *OnPaint* eseményére kell beírni.

- **OnDeactivate**: az űrlap háttérbe kerülésekor következik be
- **OnCloseQuery**: az űrlap bezárása előtt következik be. Ebben még megállítható a bezárás (lásd 5.11. ábra).
- **OnClose**: az űrlap bezárásakor következik be. Ebben az eseményjellemzőben beállíthatjuk a bezárás módját (5.11. ábra)
- **OnDestroy**: az űrlap megszüntetésekor következik be



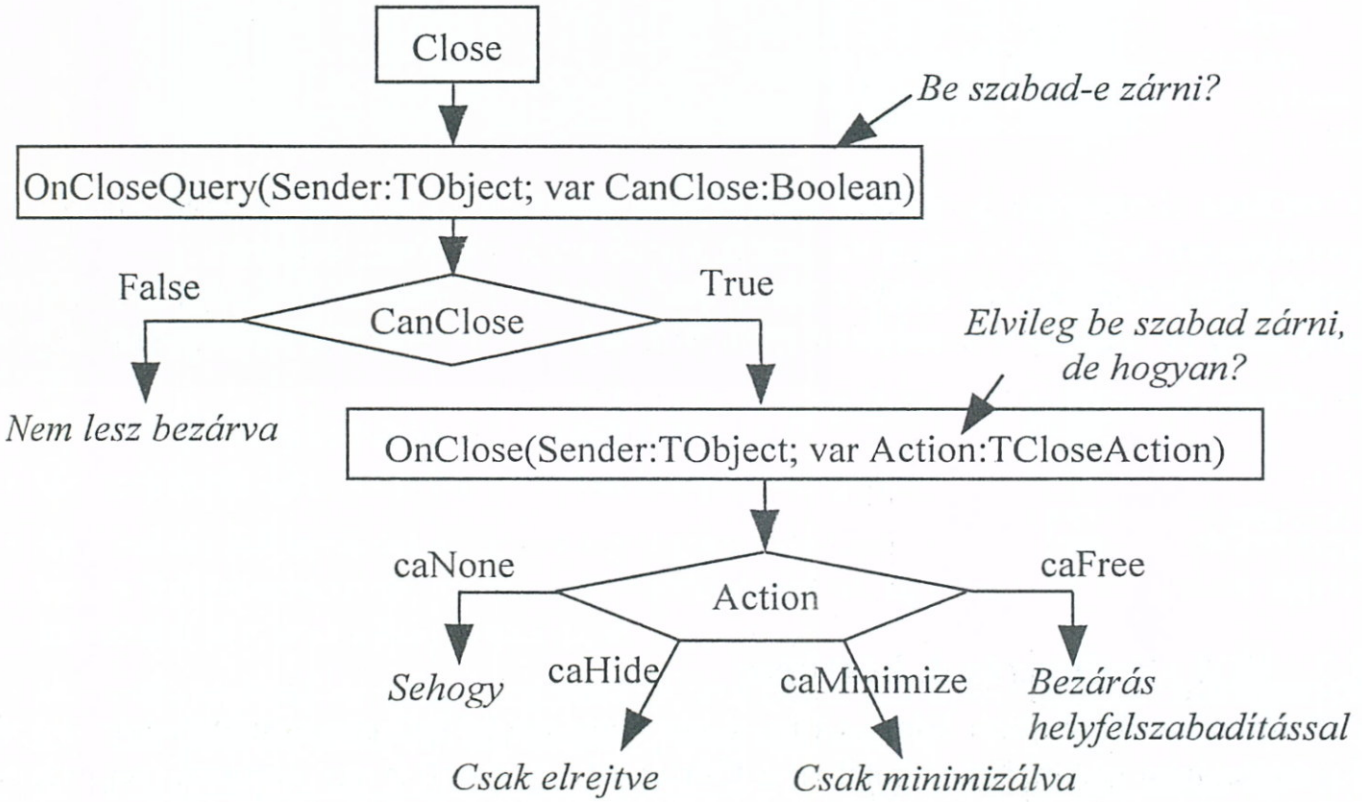
5.10. ábra. Egy űrlap életciklusa

Az események bekövetkezésének sorrendjét az 5.10. ábra szemlélteti.

Az űrlapok bezárása bonyolult folyamat. Amikor az űrlap rendszermenüjére duplán kattintunk (vagy amikor meghívjuk *Close* metódusát), elindul az 5.11 ábrán látható „láncreakció". Először is meghívódik az *OnCloseQuery* eseménybe írt kódrészlet. Ebben még felülbírálható a helyzet: meghiúsíthatjuk a bezárást (*CanClose:=False*), vagy „hagyhatjuk a maga útjára" (*CanClose:=True*; ez az alapértelmezés is). Ha nem állítottuk meg a bezárást, akkor folytatódik a „láncreakció": lefut az *OnClose*. A további teendők az *Action* címszerinti paraméter értékétől függnek:

- *Action = caNone* ⇨ az űrlap nem lesz bezárva

- *Action = caMinimize* ⇨ az űrlap minimalizált állapotba kerül
- *Action = caHide* ⇨ az űrlap háttérbe kerül
- *Action = caFree* ⇨ az űrlap bezárul, az általa lefoglalt memóriaterület felszabadul



5.11. ábra. Űrlapok bezárása

Az űrlapok bezárását nyilván a nagyobb rugalmasság kedvéért írták ilyen bonyolultra. A gyakorlatban egy leegyszerűsített mintát használunk: az *OnCloseQuery* eseményben kérdezünk rá a mentésre (vagy bezárásra, feladattól függően), az *OnClose*-ban pedig a tényleges bezárás mellett voksolunk (*Action:=caFree*).
Például egy szövegszerkesztőben, a dokumentumablak bezárása előtt ajánlatos rákérdezni a mentésre: „Kívánja menteni a változtatásokat?" A felhasználónak három lehetőséget kell felkínálnunk:

- *Igen*: megtörténik a mentés, majd bezárjuk az ablakot.
- *Nem*: nem mentünk, de az ablakot bezárjuk.
- *Mégsem*: a felhasználó meggondolta magát, az ablakot már nem akarja bezárni.

Ennek a következő a kódja:

```
procedure TfrmChild.FormCloseQuery(Sender: TObject;
                                   var CanClose: Boolean);
```

```
begin
   Case MessageDlg('Kívánja menteni a változtatásokat?',
        mtConfirmation, [mbYes, mbNo, mbCancel],0) Of
      mrYes:    begin
                   Save1Click(Sender);
                   CanClose:= True;
                end;
      mrNo:     CanClose:= True;
      mrCancel: CanClose:= False;
   end;
end;
```

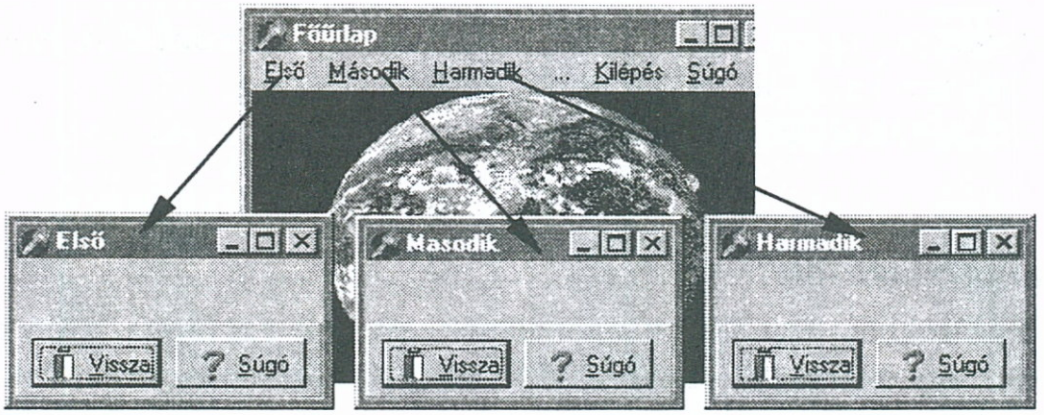
A *CanClose* címszerinti paraméter értékét nem kötelező True-ra állítanunk, mivel ez az alapértelmezett értéke. Mi mégis megtettük a jobb áttekinthetőség kedvéért.
Ha már eldöntöttük mi lesz, és eljutunk az *OnClose* eseménybe, akkor ez egyértelműen azt jelenti, hogy az űrlapot be szeretnénk zárni. Hát akkor ez történjen is ténylegesen meg!

```
procedure TfrmChild.FormClose(Sender: TObject;
                              var Action: TCloseAction);
begin
   Action:= caFree;
end;
```

5.2.7 SDI alkalmazások készítése

Ha alkalmazásunkban több űrlap is van, akkor alaposan ki kell gondolnunk ezek megjelenítési és bejárási sorrendjét. Általában a főűrlapról egy menü (vagy gombok) segítségével el lehet jutni a többi ablakra, az ablakokról pedig a *Vissza* gombbal lehet újra a főűrlapra kerülni (lásd 5.12. ábra). Természetesen az is elképzelhető, hogy bizonyos űrlapokról tovább lehet lépkedni másokra (nem a főűrlapon keresztül); ezt már a konkrét feladat logikája határozza meg. A lényeg csak az, hogy felhasználóbarát alkalmazás készüljön.

Minden űrlapon megtalálhatók bizonyos általános célú gombok: *Vissza*, *Súgó*, *Nyomtatás*... Jó szokás ezeket egy – az űrlap valamely részéhez igazított – *panel*-re helyezni. Az is a felhasználóbarátsághoz tartozik, hogy az azonos célú gombok minden ablakban ugyanott helyezkedjenek el.
Az azonos megjelenítést elősegíti a Delphi 32 bites változataiban bevezetett **vizuális űrlapörökítési mechanizmus**. Egy űrlapot (például a *Vissza* és *Súgó* gombos űrlapot) kimenthetünk az űrlapminták közé a gyorsmenü *Add to Repository* parancsával. Később, amikor egy új űrlapot szeretnénk készíteni ez alapján, akkor a *File/New*...űrlap *Forms* oldaláról válasszuk ki az előzőleg lementett mintát, majd jelöljük ki az *Inherited* választógombot. Ezzel egy olyan űrlapot hozunk létre, mely a minta-űrlaposztály leszármazottja lesz. Ha később változtatunk a mintán (például elhelyezünk rajta még egy *Nyomtatás* gombot is), akkor ennek utódjai is változni fognak. Sőt, a *Vissza* és *Súgó* gombokat már az ősben (az űrlapmintában) lekódolhatjuk, és ezt a kódot automatikusan az utódok is végre fogják hajtani. Az űrlapörökítési mechanizmussal bővebben a 10. fejezetben foglalkozunk.



5.12. ábra. Az űrlapok bejárása

Térjünk vissza az SDI alkalmazásokhoz. Hogyan lehet egy űrlapról egy másikat megjeleníteni? Hogyan léphetünk a *Főűrlapról* az *Első* űrlapra (5.12. ábra)?
Lépések:

- Megtervezzük a *főűrlapot*, menüszerkezetét ⇨ *frmMain* (*uMain* egység).
- Megtervezzük az *Első* űrlapot ⇨ *frmElso* (*uElso* egység).
- Az *frmElso* deklarációja az *uElso* egységben található. Ahhoz, hogy az *uMain* egységben is hivatkozhassunk rá, be kell szerkesztenünk egységét ⇨ az *uMain* egységbe helyezzük el a *uses uElso* hivatkozást.
- Már csak az *frmElso* űrlap megjelenítés van hátra. Ezt a főűrlap *Első* menüpontjának *OnClick* eseményében valósítjuk meg (*frmElso.Show*).

```
Unit uMain;
Interface
uses SysUtils...;

Type
   TfrmMain = Class(TForm)
   ...
   end;
var
   frmMain: TfrmMain;

Implementation
uses uElso;
Procedure TfrmMain.Elso1Click;
begin
   frmElso.Show;
end;
end.
```

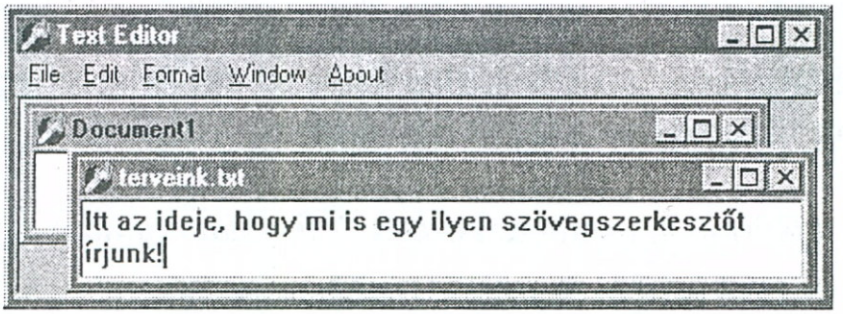
```
Unit uElso;
Interface
uses SysUtils...;

Type
   TfrmElso = Class(TForm)
   ...
   end;
var
   frmElso: TfrmElso;

Implementation

...

end.
```

5.13. ábra. SDI alkalmazások készítése

Gyakorlásképpen hozza létre az 5.12. ábrán látható alkalmazást! A súgó egyelőre elmaradhat, ezzel a 16. fejezetben ismerkedünk majd meg.
(🖫 5_URLAPOK\PURLAPOK.DPR)

5.2.8 MDI alkalmazások készítése

Az MDI alkalmazásokban van egy keretablakunk, és sok egyforma gyerekablakunk. Bemutatásul írjunk egy szövegszerkesztőt (5.14. ábra). Kezdetben egyetlen dokumentumablak se legyen nyitva.



5.14. ábra. Szövegszerkesztő

Lépések:

- Megtervezzük a keretablakot, stílusát (*FormStyle*) *fsMDIForm*-ra állítjuk ⇨ *frmMain* (*uMain* egység).
- Megtervezzük a gyerekablakot, stílusát *fsMDIChild*-ra állítjuk ⇨ *frmChild* (*uChild* egység).
- Beállítjuk alkalmazásunk űrlapjait (*Project/Options, Forms* oldal):
  - Főűrlap (*Main form*): *frmMain*
  - *Autocreate* űrlapok (*Auto-create forms*): *frmMain*
  - Egyéb űrlapok (*Available forms*): *frmChild*

  Ha az *frmChild* űrlapot „*auto-create*"-nek hagynánk, akkor a szövegszerkesztő elindításakor már eleve létezne egy nyitott dokumentum.
- A keretablak *New* menüpontjával hozzunk létre egy gyerekablakot, majd jelenítsük is meg (5.15. ábra).

  A gyerekablak egységéből töröljük ki az űrlapobjektum deklarációját. A gyerekablakot helyben fogjuk deklarálni és létrehozni a *New* menüpont kattintására. Megjelenítése előtt elvégezhetünk bizonyos beállításokat (például címsorába kiírjuk: Dokumentum1).

```
Unit uMain;
Interface
uses SysUtils...;

Type
   TfrmMain = Class(TForm)
   ...
   end;

Implementation
uses uChild;

Procedure TfrmMain.New1Click(Sender:
                              TObject);
var frmChild: TfrmChild;
begin
   frmChild:= TfrmChild.Create(self);
   With frmChild do
      begin
         {esetleges beállítások}
         Show;
      end;
end;
end.
```

```
Unit uChild;
Interface
uses SysUtils...;

Type
   TfrmChild = Class(TForm)
   ...
   end;
var
   frmChild: TfrmChild;
Implementation

...

end.
```

5.15. ábra. MDI alkalmazás készítése (Az *uChild* egységbeli űrlapobjektumot kitörölhetjük, hiszen saját lokális objektummal dolgozunk a *New1Click*-ben)

Ennyit általánosságban az MDI alkalmazásokról, most nézzük mindezt konkrétan a szövegszerkesztőnél!

5.2.9 Feladat: MDI szövegszerkesztő írása

Készítsük el az 5.14 ábrán látható szövegszerkesztőt! Egyszerre több dokumentummal is dolgozhassunk benne (mint a Wordben). Lehessen a szöveget szerkeszteni, formázni, vágólapra helyezni és vágólapról beolvasni. A megszerkesztett szöveget lehessen lementeni állományba és állományból visszatölteni.

Megoldás (🖫 5_SZOVSZ\PSZOVSZE.DPR)
Tervezzük meg a főűrlapot!

Komponens	Jellemző	Érték
frmMain: TfrmMain	Caption	'TextEditor'
	formStyle	fsMDIForm

Milyen legyen a menüszerkezet? Elvileg a szövegszerkesztőben elérhető funkciók a következők lennének:

File	Edit	Format	Window	About
New	Cut Ctrl+X	Character	Tile	
Open...	Copy Ctrl+C		Cascade	
Save	Paste Ctrl+V		Arrange icons	
Save as...	Select all		*nyitott gyerekablakok listája*	
Close				
Exit				

GroupIndex=0 *GroupIndex=3* *GroupIndex=4*

File	Window	About
New	Tile	
Open...	Cascade	
Exit	Arrange icons	
	nyitott gyerekablakok listája	

← *keretablak menüje*

fölülírja a keret-ablak File menüjét

beszúródnak a File és Window közé

GroupIndex=0 *GroupIndex=1* *GroupIndex=2*

File	Edit	Format
New	Cut Ctrl+X	Character
Open...	Copy Ctrl+C	
Save	Paste Ctrl+V	
Save as...	Select all	
Close		
Exit		

← *gyerekablak menüje*

5.16. ábra. A keret- és gyerekablakok menüszerkezetének összefésülése

A kérdés az, hogy mely menüpontokat helyezzük a keretablakba, és melyeket a gyerekablakba? Minden menüpontot abba az űrlapba kell elhelyezni, amelyikre értelmezve van. Az aláhúzott menüpontok az aktuális gyerekablakra vonatkoznak, így ezeket majd a gyerekablak mintába fogjuk implementálni. Kezdetben, amikor még nincs egy gyerekablak

sem, akkor még csak a keretablak menüje látható, később viszont (ha már van legalább egy gyerekablak) a keret- és gyerekablak menüi összefésülődnek. De vajon hogyan? Erre kapunk választ az 5.16. ábrán. A keret- és gyerekablak főmenüpontjai a *GroupIndex* növekvő sorrendjében helyezkednek el, ha pedig két menüpontnál ez az érték azonos (lásd *File* menü), akkor itt a gyerekablak menüpontja fog látszani.

A menük összefésülése csak a főmenüpontokra érvényes. Nem tudjuk például a *File* menüt összerakni a keret- és a gyerekablak almenüpontjaiból: vagy a keret- vagy a gyerekablak *File* menüje lesz a „nyertes". Ezért kénytelenek vagyunk megismételni a gyerekablakban a *New*, *Open* és *Exit* menüpontokat, habár ezek logikailag csak a keretablakba tartoznának.

Visszatérve a keretablakhoz, hozzunk le rá egy *TMainMenu* komponenst, és tervezzük meg az 5.16. ábra szerint. A nyitott gyerekablakok listáját szerencsére nem nekünk kell beprogramoznunk. Elég beállítanunk a keretablak *WindowMenu* jellemzőjét a *Window* menüpontra: *frmMain.WindowMenu:= Window1* (*Window1* a *Window* menüpont alapértelmezés szerinti neve).

Emlékeztetőül!

- Egy menüpontban az aláhúzott karaktert a *Caption* jellemzőjébe írt '&' karakter jelzi (például *E&xit* ⇨ *Exit*).
- Szeparátort a *Caption*-ba írt '-' jellel hozhatunk létre.
- A *GroupIndex* jellemzőt csak a főmenüpontoknál kell beállítanunk.

- Tervezzük meg a gyerekablakot (*frmChild*)! A szöveget egy többsoros szerkesztődobozban fogjuk szerkeszteni.

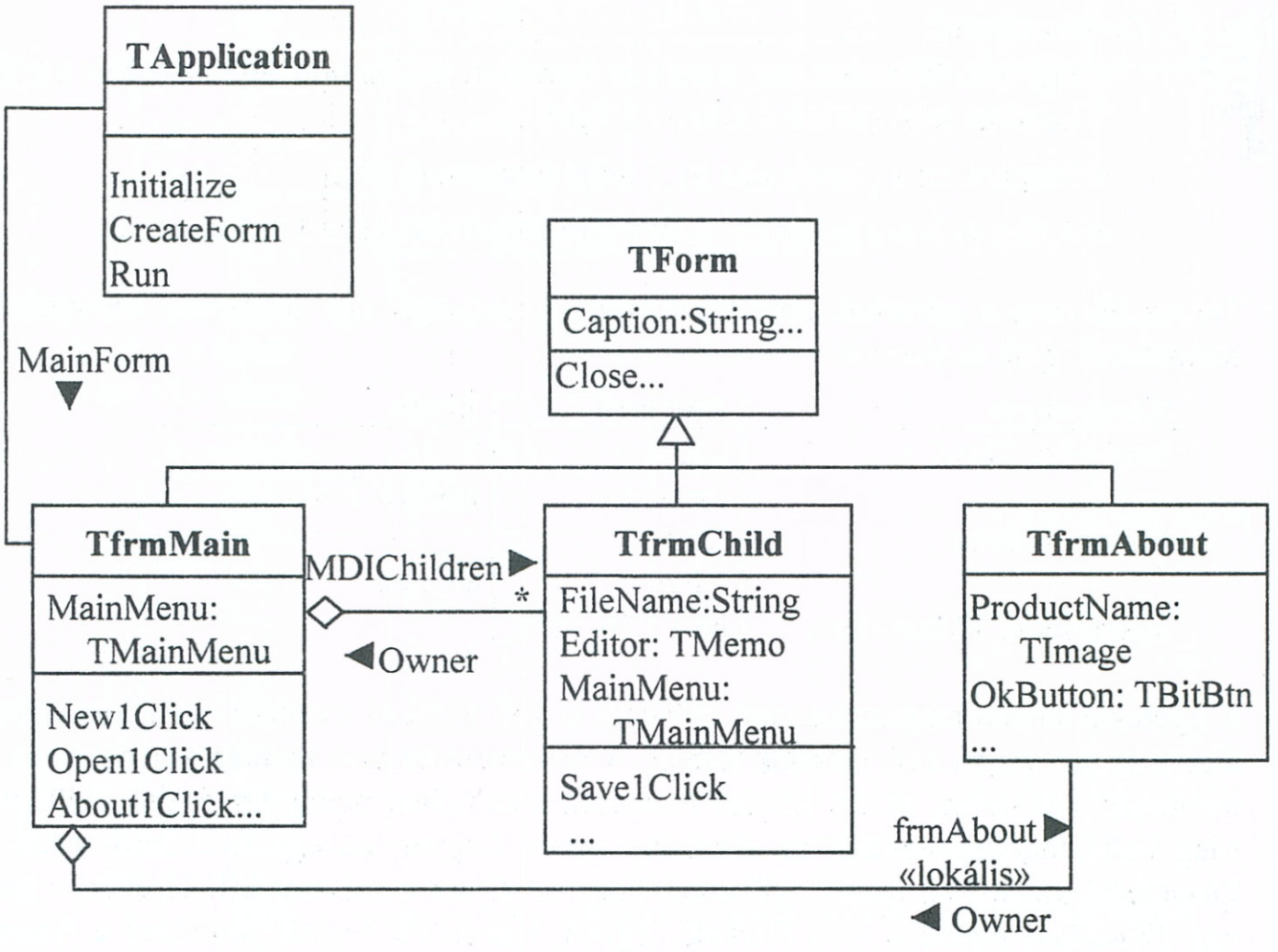
Komponens	**Jellemző**	**Érték**
frmChild: TfrmChild	Caption	''
	formStyle	fsMDIChild
Editor: TMemo	Lines	''
	Align	alClient
MainMenu: TMainMenu	lásd 5.16. ábra	

A gyerekablakban megszerkesztett szöveget ki kell tudnunk menteni egy állományba, illetve onnan vissza kell tudnunk tölteni. A dokumentum első mentésekor bekérjük a mentendő fájl nevét, az ezt követő mentések alkalmával pedig ezt fogjuk felülírni. Meg kell tehát jegyeznünk az állomány nevét. Legyen ez egy nyilvános mező a *TfrmChild* osztályban (azért nyilvános, hogy az *Open* menüpont kifejtésében – ami egy *TfrmMain* metódus – is elérhető legyen, hiszen ott fogjuk beállítani a frissen megnyitott állomány nevére):

```
unit uChild;
...
Type
  TfrmChild = Class(TForm)
     ...
  public
     FileName:String;
  end;
```

- Hozzuk létre a névjegyablakot (*frmAbout*)!

  Tipp: használjuk az *AboutBox* űrlapmintát (a *File/New, Forms* oldalról).
- Következő lépésünk: az alkalmazás űrlapjainak beállítása (*Project/Options*):
  - Főűrlap: *frmMain*
  - *Auto-create* űrlapok: *frmMain*
  - Egyéb űrlapok: *frmChild, frmAbout*
- Kódolás előtt készítsük el a szövegszerkesztőnk „kiforrott" osztálydiagramját:



5.17. ábra. A szövegszerkesztő osztálydiagramja

Tehát:

- Az alkalmazás „fő motorja" az *Application:TApplication* objektum. Ez létrehozza, majd megjeleníti a főablakot (*CreateForm*), azután pedig elindítja az üzenetkezelő ciklust (*Application.Run*). Erős tartalmazási kapcsolatban áll a főablakkal a *MainForm = frmMain* mezőjén keresztül.
- Minden űrlapunk osztálya a keretrendszerbe beépített *TForm* osztály leszármazottja. Számos adatot és metódust örökölnek innen (közöttük a *Close*). Az új űrlaposztályokban csak a sajátosságok vannak feltüntetve.
- A főablak (*frmMain*) MDI keretablak típusú, tehát futási időben lehet akárhány gyerekablaka (ezt jelzi az osztálydiagramon a '*'). A főablak tehát sok gyerekablakkal áll kapcsolatban. Az **egy-sok** kapcsolatot ún. konténer objektumok segítségével szokás megvalósítani. Ezt tapasztaljuk itt is: a gyerekablakok az *MDIChildren* listára vannak felfűzve.
- Az előbb említett kapcsolat kétirányú: a főablak ismeri a gyerekablakokat (*Tfrmain.MDIChildren*), de a gyerekablakok is ismerik a főablakot, hiszen ő a tulajdonosuk (*TfrmChild.Owner*).
- A főablakból megjeleníthető a névjegyablak (*frmAbout*). Ez a *TfrmAbout* űrlaposztály példányaként jön létre. A főablak és a névjegyablak közötti kapcsolat lokális, hiszen csak a *TfrmMain.About1Click* metódus futásának ideje alatt érvényes.

- Lépjünk vissza a főablakba, és kódoljuk le menüpontjait:

```
...
Const DefaultCaption = 'Document'; {ez lesz kezdetben a dokumentu-
mok címe}
...

procedure TfrmMain.New1Click(Sender: TObject);
var Child: TfrmChild;
begin
  Child:= TfrmChild.Create(self);
  With Child Do
     Begin
        Caption:= DefaultCaption + IntToStr(self.MDIChildCount);
        Show;
     End;
end;

procedure TfrmMain.Exit1Click(Sender: TObject);
begin
  Close;
end;
```

Ha nem minősítünk Self-fel, akkor a gyerekablak (Child) gyerekeinek számát veszi, ez pedig 0 lesz.

```
procedure TfrmMain.Tile1Click(Sender: TObject);
begin
  Tile;
end;

procedure TfrmMain.Cascade1Click(Sender: TObject);
begin
  Cascade;
end;

procedure TfrmMain.ArrangeIcons1Click(Sender: TObject);
begin
  ArrangeIcons;
end;

procedure TfrmMain.About1Click(Sender: TObject);
Var frmAbout:TfrmAbout;
begin
  frmAbout:= TfrmAbout.Create(self);
  With frmAbout Do
     Begin
        ShowModal;
        Free;{bezárás után felszabadítjuk a memóriát}
     End;
end;
```

Az állományok megnyitásánál szükség van egy *OpenDialog* komponensre. Helyezzünk el egyet a főablakon, majd folytassuk a kódolást az *Open* menüponttal:

```
procedure TfrmMain.Open1Click(Sender: TObject);
Var Child:TfrmChild;
begin
  if OpenDialog.Execute then
     Begin
        Child:= TfrmChild.Create(self);
        With Child Do
           Begin
              {tároljuk az állomány nevét}
              FileName:= OpenDialog.FileName;
              {címében elhelyezzük az áll. nevét, levágva az
               elérési útvonalat}
              Caption:= ExtractFileName(FileName);
              Editor.Lines.LoadFromFile(FileName);
              {most töltöttük be, tehát még nincs változás}
              Editor.Modified:= False;
              Show;
           End;
     End;
end;
```

Futtassa le így az alkalmazást! Mit vesz észre? A gyerekablakokat nem lehet bezárni, csak minimizálni.

- Tegyük bezárhatóvá a gyerekablakot!

```
procedure TfrmChild.FormClose(Sender: TObject;
                              var Action: TCloseAction);
begin
  Action:= caFree;
end;
```

- Kódoljuk le a gyerekablak menüpontjait is! Természetesen ide is el kell helyeznünk egy *SaveDialog* és egy *FontDialog* komponenst. A *New*, *Open* és *Exit* menüpontokat már egyszer lekódoltuk a keretablakban (⇨ *TfrmMain.New1Click, TfmrMain. Open1Click és TfrmMain.Exit1Click*); ezeket a metódusokat fogjuk a gyerekablakból meghívni (természetesen ehhez „látnunk" kell az *uMain* egységben deklaráltakat ⇨ *uses uMain*).

```
procedure TfrmChild.New1Click(Sender: TObject);
begin
  frmMain.New1Click(Sender);
end;

procedure TfrmChild.Open1Click(Sender: TObject);
begin
  frmMain.Open1Click(Sender);
end;

procedure TfrmChild.Close1Click(Sender: TObject);
begin
  Close;
end;

procedure TfrmChild.Save1Click(Sender: TObject);
begin
  If Copy(Caption, 1,length(DefaultCaption)) = DefaultCaption
  Then
     SaveAs1Click(Sender) {ha még nem volt soha elmentve}
  Else {van már neve}
     Begin
        Editor.Lines.SaveToFile(FileName);
        Editor.Modified:=False;
     End;
end;
```

```
procedure TfrmChild.Exit1Click(Sender: TObject);
begin
  frmMain.Exit1Click(Sender);
end;

procedure TfrmChild.SaveAs1Click(Sender: TObject);
begin
  If SaveDialog.Execute Then
     Begin
        FileName:= SaveDialog.FileName;
        Caption:= ExtractFileName(FileName);
        Editor.Lines.SaveToFile(FileName);
        Editor.Modified:= False;
     End;
end;

procedure TfrmChild.Cut1Click(Sender: TObject);
begin
  Editor.CutToClipboard;
end;

procedure TfrmChild.Copy1Click(Sender: TObject);
begin
  Editor.CopyToClipboard;
end;

procedure TfrmChild.Paste1Click(Sender: TObject);
begin
  Editor.PasteFromClipboard;
end;

procedure TfrmChild.SelectAll1Click(Sender: TObject);
begin
  Editor.SelectAll;
end;

procedure TfrmChild.Character1Click(Sender: TObject);
begin
  {a szöveg típusának beállítása}
  FontDialog.Font:= Editor.Font;
  If FontDialog.Execute Then
     Editor.Font:= FontDialog.Font;
end;
```

Próbálja ki az alkalmazást! Már (remélhetőleg) csak az a probléma vele, hogy bezáráskor nem kérdez rá a mentésre. Oldjuk meg ezt is!

```
procedure TfrmChild.FormCloseQuery(Sender: TObject;
                                   var CanClose: Boolean);
begin
  If Editor.Modified then
     {ha volt változtatás az utolsó mentés óta}
     Case MessageDlg('Save changes?', mtConfirmation,
                     [mbYes, mbNo, mbCancel],0) Of
       mrYes:  Begin
                  Save1Click(Sender);
                  CanClose:= True;
               End;
       mrNo:   CanClose:= True;
       mrCancel:CanClose:= False;
     End;
end;
```

- Ha igazán „széppé" akarjuk tenni alkalmazásunkat, akkor oldjuk meg a menüpontok szürkítését is: minden menüpont csak akkor legyen elérhető, ha ténylegesen van értelme. Az egyes menüpontok elérhetőségének vizsgálatát a főmenüpontok kattintásának eseményébe szoktuk írni. Vegyük sorba a főmenüpontokat:
  - *File*: almenüpontjai közül a *Save, Save As...* és a *Close* menüpontok „problémásak", de ezek – a menüpontok összefésülése miatt – eleve csak akkor láthatók, ha van legalább egy gyerekablak.
  - *Edit*: a *Cut* és *Copy* menüpontok csak akkor értelmezhetők, ha a dokumentumban valami ki van jelölve, a *Paste* pedig csak akkor, ha a vágólapon szöveges információ található (képet nem tudunk *Memo*-ban elhelyezni). A *SelectAll* mindig kiválasztható.
  - *Format*: a karakterformázás mindig érvényes.
  - *Window*: almenüpontjainak csak akkor van értelme, amikor van legalább egy gyerekablak.
  - *About*: mindig elérhető.

Kódot tehát csak az *Edit* és a *Window* menüpontokra kell írni:

```
{az uChild egységbe}
procedure TfrmChild.Edit1Click(Sender: TObject);
begin
  Cut1.Enabled:= Editor.SelLength>0;
  Copy1.Enabled:= Editor.SelLength>0;
  Paste1.Enabled:= Clipboard.HasFormat(CF_Text);
end;
```

A Clipboard objektum a Clipbrd egységben van deklarálva ⇨ Uses Clipbrd

```
{az uMain egységbe}
procedure TfrmMain.Window1Click(Sender: TObject);
Var En:Boolean;
begin
  En:= MDIChildCount>0;
  Tile1.Enabled:=En;
  Cascade1.Enabled:=En;
  ArrangeIcons1.Enabled:=En;
end;
```

Feladatok

Egészítse ki a „Csupafül" alkalmazást (5.2. ábra) a következő menüvel. A „pipa" mindig az aktuális oldal címe mellett legyen látható. A névjegy menüpont hatására nyíljon meg egy névjegy űrlap.

Felsőfülek	Alsófülek	Névjegy	Kilépés
✓ Első Második Harmadik	✓ Álló kép Animáció		

Tekintse át a DELPHI\DEMOS\DOC\FILMANEX\ mintapéldát, mely egy kisebb fájlkezelő programot tartalmaz. Alakítsa át (vagy még jobb lenne, ha elölről elkészítené) úgy, hogy többablakos MDI alkalmazás legyen (akárcsak a Win 3.1-es fájlkezelője), és az ablakok között lehessen vonszolással másolni és áthelyezni az állományokat.

6. Grafika, nyomtatás

Delphiben bizonyos grafikai elemeket már tervezési időben az űrlapunkra „varázsolhatunk", másokat pedig csak futáskor. Ebben a fejezetben a Delphi grafikai lehetőségeit fogjuk előbb megvizsgálni (6.1. és 6.2. pontok), majd be is gyakorolni feladatokon keresztül (6.3. pont). A 6.4. pontban áttekintjük a nyomtatással kapcsolatos tudnivalókat.

6.1 Tervezési időben létrehozható grafikai elemek

A *TBitBtn*, *TSpeedButton*, *TShape* és *TImage* komponensek lehetővé teszik bizonyos grafikai elemek megjelenítését már tervezési időben. A *TBitBtn-ról* és a *TSpeedButton*-ról már a 4. fejezetben beszéltünk, most nézzük a többit:

6.1. ábra. Grafika megjelenítése tervezési időben

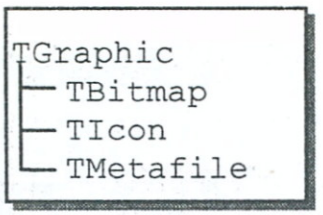
6.1.1 TShape (Additional paletta)

- Helye az osztályhierarchiában: *TObject/TComponent/TControl/...TShape*
- Szerepe: ellipszis, kör, téglalap vagy négyzet alakú síkidom megjelenítése
- Fontosabb jellemzői:
  - **Shape**: az alakzat formája. Lehetséges értékei: *stEllipse, stCircle, stRectangle, stRoundRect, stSquare, stRoundSquare*.

- **Pen: TPen**
  Tartalmazza a rajzoló toll tulajdonságait (színét, vastagságát, stílusát...). A *TPen* osztályt a 6.2.1. pontban ismertetjük.
- **Brush: TBrush**
  Tartalmazza az idomokat kitöltő ecset tulajdonságait (színét, stílusát...). A *TBrush* osztályt lásd a 6.2.1. pontban.

6.1.2 TImage (Additional paletta)

- Helye az osztályhierarchiában: *TObject/TComponent/TControl/...TImage*
- Szerepe: egy BMP, WMF vagy ICO formátumú grafika megjelenítése
- Jellemzői:
  - **Picture: TPicture**
    Objektum, mely tárolja a megjelenített grafika adatait: a kép méreteit a **Width** és **Height** mezőiben, magát a grafikát pedig a **Graphic: TGraphic** mezőjében. A *TGraphic* a grafikus osztályok közös őse (lásd 6.2. ábra). Egy adott pillanatban tehát az *Image.Picture.Graphic* vagy egy bittérképet tartalmaz (*TBitmap*), vagy egy ikont (*TIcon*), vagy egy windows metafájl képet (*TMetafile*).



6.2. ábra. Grafikus osztályok a Delphiben

A *Picture* jellemző értékét beállíthatjuk akár tervezési-, akár futási időben: tervezéskor egy képbetöltési párbeszédablak segítségével, futáskor pedig a **TPicture** osztály **LoadFromFile** metódusával.
Például:

```
Image.Picture.LoadFromFile('EARTH.BMP');
vagy
Image.Picture.LoadfromFile('SKYLINE.ICO');
vagy
Image.Picture.LoadFromFile('ARTIST.WMF');
```

A *LoadFromFile* egy polimorf metódus. A betöltendő állomány kiterjesztéséből „kitalálja" a képformátumot, így az annak megfelelő szabályok alapján tölti azt be, majd helyezi el a *Image.Picture.Graphic* mezőbe. A betöltött kép méretei az *Image.Picture.Width* és *Image.Picture.Height* mezőkbe kerülnek.

Később meglátjuk, hogy egy *TImage* komponensre futás közben akár egérrel is lehet rajzolni, így felvetődhet a rajz elmentésének gondolata. Ezt a műveletet a **TPicture** osztály **SaveToFile** metódusával valósíthatjuk meg.

- **Stretch:Boolean**
  A *Stretch* jellemző Igaz értéke esetén a kép felveszi a *TImage* komponens méreteit (Figyelem, ez torzítással járhat!) Hamis érték esetén a kép a komponens balfelső sarkához igazítva eredeti méreteiben jelenik meg.

6.2 Futási időben létrehozható grafikai elemek

Delphiben azoknak a komponenseknek a felületére rajzolhatunk, amelyek rendelkeznek *Canvas* jellemzővel. Ez az illető komponens rajzvászon-objektuma, mely a rajzoláshoz szükséges tulajdonságokat (toll, ecset) és metódusokat (*LineTo, PolyLine...*) tartalmazza. A *Canvas* egy nyilvános (*public*) jellemző, azaz csak futás közben érhető el, tervezési időben még nem. Emiatt a rajzolást kódból kell megvalósítanunk. Nézzük milyen eszközök állnak rendelkezésünkre!

6.2.1 TCanvas osztály

- Szerepe: ez a komponensek rajzvásznának az osztálya, mely tartalmazza a rajzeszközök (toll, ecset, betűtípus) és a rajzvászon tulajdonságait. Metódusainak segítségével rajzolhatunk a vászonra.
- Fontosabb jellemzői:
  - **Pen: TPen** tartalmazza a rajzoló toll adatait:
    - **Color: TColor** (4 bájtos egész szám)
      A toll színe. Lehetséges értékei: *clRed*, *clWhite*, *clGreen*... előredefiniált konstansok, de ugyanakkor „keverhetünk" saját színeket is. Persze ehhez ismernünk kell a négy byte értelmezését:

4. bájt	3. bájt	2. bájt	1. bájt
nem használt	kék	zöld	piros

      A színt megadhatjuk szám formájában vagy az **RGB** függvény segítségével. Az RGB függvény a három paraméteréből (**R**ed = piros, **G**reen = zöld és **B**lue = kék) előállítja a megfelelő színt. Például:
      $00FF0000 = RGB(0,0,$FF) = kék
      $00FFFFFF = RGB ($FF,$FF,$FF) = fehér
      $00000000 = RGB(0,0,0) = fekete...

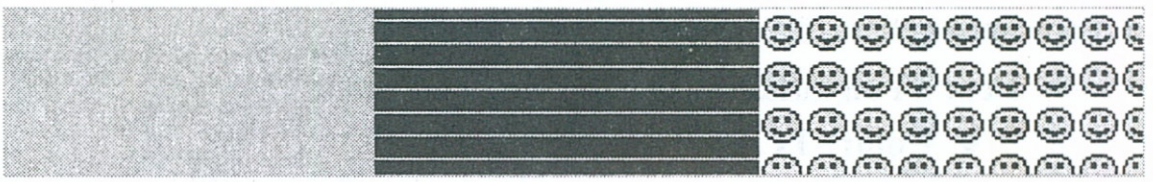
    - **Style: TPenStyle**
      A toll stílusa. Lehetséges értékei: *psSolid* (folytonos) , *psDash* (szaggatott), *psDot* (pontozott)...

- **Width: Integer**
  A toll vastagsága. Alapértelmezés szerint 1 pixeles. Figyelem, a 2 pixeles vagy az ennél is vastagabb tollakkal már csak folytonos vonal rajzolható.
- **Mode: TPenMode**
  Rajzmód, mely meghatározza a toll színének érvényesülési módját. Lehetséges értékei: *pmCopy* ⇨ a toll felülírja a hátteret, *pmXOR* ⇨ a toll és a háttérszín között XOR (kizáró vagy) műveletet hajt végre, *pmNotXOR*, *pmBlack*... (Alkalmazását lásd a 6.3.1. pontban.)

Brush: TBrush

Tartalmazza a zárt idomok kitöltésére használt ecset adatait:

- **Color: TColor**
  Az ecset színe. A 6.3.ábra esetében ez szürke (*clSilver*).
- **Style: TBrushStyle**
  Az ecset stílusa. Lehetséges értékei: *bsSolid* = teljes kitöltés (6.3.ábra első kép), *bsHorizontal* = az ecset színével vízszintes csíkminta fekete háttérben (6.3.ábra második kép)...
- **Bitmap: TBitmap**
  Ha ezt a jellemzőt beállítjuk egy 8x8-as bittérképre, akkor a zárt idomok kitöltése ilyen mintázattal fog megtörténni. Ez esetben az ecset színe és stílusa nem számít (6.3.ábra harmadik kép).



6.3.ábra. Ecsetek

Font: TFont

Tartalmazza a rajzvásznon megjelenítendő betűk jellemzőit:

- **Name**: a betűtípus neve
- **Size**: a betűméret
- **Color**: a betű színe
- **Style**: a betű stílusa, kiemeltsége. Ez egy halmaz a következő lehetséges elemekkel: *fsBold, fsItalic, fsUndeline, fsStrikeOut.*

Például:
```
Name = 'Arial CE'
Size = 14
Color = clBlack
Syle = [fsBold, fsItalic, fsUnderline]
```

Szöveg

- **ClipRect:TRect**
  A rajzvászon határai. A határokon kívülre eső rész „le lesz vágva".
  Bal felső sarka: (*Canvas.ClipRect.Left, Canvas.ClipRect.Top*)
  Jobb alsó sarka: (*Canvas.ClipRect.Right, Canvas.ClipRect.Bottom*)
- **Pixels[X,Y:Integer]: TColor**
  A vászon pixelenkénti színinformációja. Elvileg a *Pixels* tömb elemeinek beállításával is rajzolhatunk a vászonra, azonban, ha lehet, akkor inkább az erre bevezetett metódusokat használjuk.

• Fontosabb metódusai:

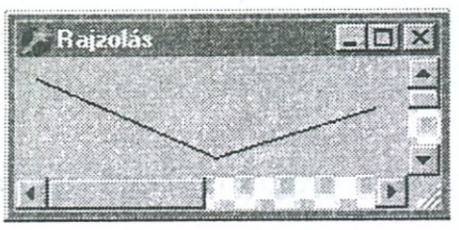
Rajzolás az aktuális tollal:

- **MoveTo(X,Y:Integer)**
  Kurzor pozícionálása az (X,Y) pontba. (Rajzolás nem történik!)
- **LineTo(X,Y:Integer)**
  Vonal húzása az aktuális kurzorpozícióból az (X,Y) pontba. A kurzor átkerül az (X,Y) pozícióba.
- **Polyline(Pontok: Array of TPoint)**
  Nyitott sokszög rajzolása a paraméterként megadott ponttömb pontjainak összekötésével. (Zárt sokszög rajzolásához lásd később a *Polygon* metódust.)
  Például a két alábbi programrészlet hatása azonos.

```
With frmRajzolas.Canvas Do
  Begin
    MoveTo(10,10);
    Lineto(100,50);
    LineTo(180,25);
  End;
```

≡

```
With frmRajzolas.Canvas Do
  Polyline([Point(10,10),
  Point(100,50),  Point(180,25)])
```



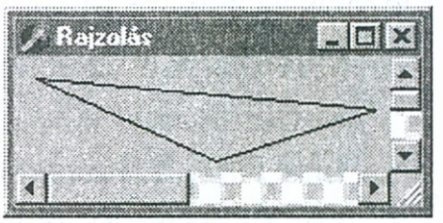
6.4. ábra. Nyitott sokszög rajzolása

Rajzolás az aktuális tollal, kitöltés az aktuális ecsettel (Csak a gyakoribb metódusokat mutatjuk be):

- **Fillrect(ARect: TRect)**
  Paraméterként megadott téglalapnyi terület kifestése az aktuális ecsettel.
- **Rectangle(X1, Y1, X2, Y2:Integer)**
  Paraméterként megadott bal-felső és jobb-alsó sarkú téglalap rajzolása. Annyiban különbözik a *Fillrect* metódustól, hogy ez még kontúrt is rajzol az aktuális tollal.

- **RoundRect(X1, Y1, X2, Y2, KerX, KerY:Integer)**
  Lekerekített sarkú téglalap rajzolása. *KerX*, *KerY* a kerekítés nagyságát adják meg x, illetve y irányban.
- **Ellipse(X1, Y1, X2, Y2:Integer)**
  Az (X1,Y1) bal-felső és (X2,Y2) jobb-alsó sarkú téglalapba írt ellipszis rajzolása. Négyzet esetén az ellipszisből kör lesz.
- **Polygon(Pontok: Array of TPoint)**
  Zárt sokszög rajzolása. A *Polygon* metódus – a *Polyline*-nal ellentétben – bezárja a sokszöget, azaz a ponttömb utolsó pontját összeköti az elsővel (lásd 6.5. ábra).

```
With frmRajzolas.Canvas Do
   Polygon([Point(10,10),
            Point(100,50),
            Point(180,25)])
```



6.5. ábra. Zárt sokszög rajzolása

Szöveg megjelenítése:

- **TextOut(X,Y:Integer; Text:String)**
  Szöveg írása az adott pozícióból az aktuális *Font*-tal.

Grafika (bittérkép, windows metafájl, ikon) megjelenítése:

- **Draw(X,Y:Integer; Graphic: TGraphic)**
  Grafika megjelenítése az adott (X,Y) koordinátában (ez lesz a bal felső sarka).
- **StretchDraw(Rect:TRect; Graphic: TGraphic)**
  Grafika megjelenítése az adott téglalapnyi területen. Figyelem, a kép eltorzul, felveszi a téglalap méreteit!

A következő programrészlet egy űrlap felületére rajzol:

```
procedure TfrmRajzolas.btnRajzClick(Sender:TObject);
var B:TBitmap;
begin
  B:= TBitmap.Create;
  B.LoadFromFile('SMILE.BMP');
  Canvas.Draw(10,10,B);
  Canvas.StretchDraw(Rect(50,10,100,110),B);
  B.Free;
end;
```

6.6. ábra. *Draw* és *StretchDraw*

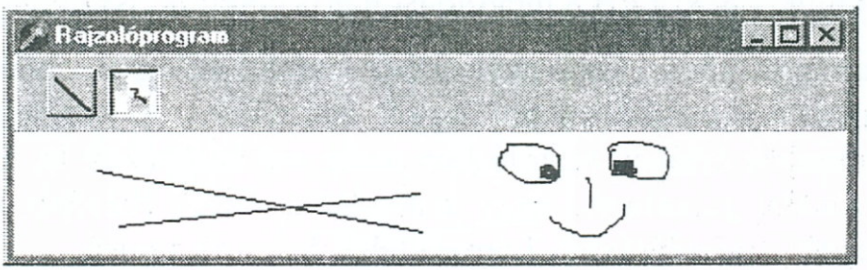
Ha az űrlapot minimizáljuk, majd utána felnagyítjuk, akkor azt tapasztaljuk, hogy rajzaink eltűnnek. Ennek oka az, hogy jelenleg a rajzolás a *btnRajz* gomb kattintására van beépítve, ez az esemény pedig átméretezésnél nem következik be. Keresni kell tehát egy olyan eseményt, mely az űrlap minden – átméretezés vagy átfedés miatti – újrarajzolásakor bekövetkezik. Ez az *OnPaint* esemény. Írja át az egész rajzolást a *TfrmRajzolas.OnPaint* eseményébe! Mit tapasztal?

6.3 Feladatok

Gyakoroljuk be az eddigi ismereteinket néhány feladaton keresztül!

6.3.1 Rajzolóprogram

Készítsünk egy rajzolóprogramot. Lehessen az egér segítségével szakaszokat húzni és szabad kézzel rajzolni. A vonalak rajzolásánál az egér bal gombjának lenyomásakor megkezdjük a rajzolást, vonszolásra „húzzuk-nyúzzuk" a szakaszt, majd felengedésre véglegesítjük.



6.7. ábra. Rajzolóprogram

Megoldás (🖫 6_RAJZOLO\PRAJZOLO.DPR)

Tervezzük meg az űrlapot (*frmRajzolo*).

Komponens	Jellemző	Érték
Panel: TPanel	Caption	''
a Panel-en: sbLine:TSpeedButton	Glyph	LINE.BMP
	GroupIndex	1
sbFreehand:TSpeedButton	Glyph	FREEHAND.BMP
	GroupIndex	1
	Down	True
Kep:TImage		*{erre fogunk rajzolni}*

Helyezzünk el űrlapunkon egy *Panel*-t (ezen lesz az eszközsor) és egy *Image* komponenst. Nem rajzolunk közvetlenül az űrlapra, hiszen akkor rajzaink ablakunk átméretezésénél

eltűnnének (hacsak nem iktatjuk be a rajzolást az űrlap *OnPaint* eseményébe is, ekkor viszont tárolnunk kell a már megrajzolt vonalak és görbék pontos helyét). Ha viszont egy *Image* komponens rajzvásznára rajzolunk, akkor elég egyszer (az egér megfelelő eseményében) megrajzolni a vonalat vagy görbét, ez bekerül a képkomponens grafikájába és mindvégig ott is marad, nem tűnik el újrarajzoláskor. Az *Image* komponensen egy fehér hátterű bittérképet fogunk létrehozni, hogy majd annak vásznára rajzoljunk.

A gyorsgomboknak kölcsönösen ki kell egymást zárniuk, emiatt állítjuk be *GroupIndex* jellemzőjüket azonos értékre. Egyiküknek a program elején már aktívnak kell lennie; legyen ez a „szabad rajzolású" gomb (*sbFreehand*). A benyomott gomb fogja tehát jelezni, hogy mi az, amit az elkövetkezőkben rajzolni fogunk. A rajzolás ténylegesen csak az egéreseményekre fog majd bekövetkezni. Honnan fogjuk akkor tudni, hogy mit kell rajzolnunk? Több megoldás közül választhatunk:

1. Közvetlenül a rajzolás előtt kérdezzük le, hogy melyik gomb van benyomva. Elemezzük kicsit ezt a megoldást: a benyomott gomb megtalálása azt feltételezi, hogy végignézzük a gombok *Down* jellemzőjét: az van benyomva, amelyiknél ez *Igaz* értékű. Két gomb esetén még „elviselhető" lenne ez a megoldás, de mi van akkor, ha 15 gombbal kell ugyanezt tennünk? Sőt, ha belegondolunk, hogy minden egér mozgatásra ezt „végig kell játszanunk", akkor ezt a megoldást máris elvetjük. Keressünk valami jobbat!
2. Vezessünk be egy *DrawingTool* nevű privát mezőt! (Elég a privát láthatóság, hiszen csak az űrlaposztály metódusaiban fogjuk használni.) A gyorsgombok kattintására beállítjuk új mezőnk értékét, rajzoláskor pedig ezt kérdezzük le. Egyelőre két értéke lehet: vagy vonalat rajzolunk, vagy szabadkézi rajzot (később ki lehet egészíteni körök, téglalapok... rajzolásával). Legyen tehát felsorolt típusú.

```
Type
   TDrawingTool = (dtLine, dtFreehand);
   TfrmRajzolo = class(TForm)
         ...
      private
         DrawingTool: TDrawingTool;
         InDraw:Boolean; //erre később lesz szükség
         ...
   end;
```

Mindkét gyorsgomb ugyanazt az eseménykezelőt hívja, ez a *DrawingToolClick*. Ahhoz, hogy könnyedén megállapíthassuk, melyik gombra kattintottak, állítsuk be (még az objektum-felügyelőben) ezek *Tag*[1] jellemzőjét (az elsőjét 0-ra, a másodikét 1-re). A *DrawingTool* metódusban ezt a jellemzőt fogjuk megvizsgálni:

---

[1] A legtöbb komponens rendelkezik egy *Tag* jellemzővel (ez nem csak Delphiben van így); ezt a rendszer nem használja, kizárólag a felhasználó számára vezették be. Bármilyen célra felhasználható, mely megkönnyíti a programozást.

```
procedure TfrmRajzolo.DrawingToolClick(Sender: TObject);
const DT: Array[0..1] of TDrawingTool = (dtLine, dtFreeHand);
begin
  DrawingTool := DT[(Sender as TComponent).Tag];
end;
```

Még rajzolás előtt – valamikor a program legelején – létre kell hoznunk egy üres bittérképet, erre fogunk majd az egérrel rajzolni.

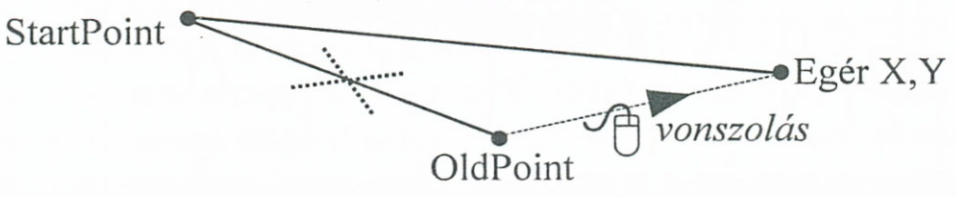
```
procedure TfrmRajzolo.FormCreate(Sender: TObject);
var B:TBitmap;
begin
  B:= TBitmap.Create;
  {A szerkesztendő kép mérete megegyezik az űrlap kezdeti méretével.}
  B.Width:= Clientwidth;
  B.Height:=Clientheight;
  Kep.Picture.Graphic:=B;
  DrawingTool:= dtFreeHand; //Szabadkézi rajzzal kezdünk
end;
```

Nézzük most **a rajzolást**! A szabadkézi rajz „nagyító" alatt apró vonalkákból áll (6.8. ábra): a töréspontok az egér mozgatásakori koordinátáiból származnak. Minden egérmozgatásra elég tehát összekötni az aktuális kurzorpozíciót – ahol az előző rajzolás után maradt – az egér pillanatnyi koordinátáival (lásd következő oldalon a kódban).

6.8. ábra. Szabadkézi rajz „nagyító" alatt

A szakaszok rajzolása már kicsit bonyolultabb. Az egér mozgatásakor még csak „igazítgatjuk" a vonalat, a végleges formája csak az egérgomb felengedésekor születik meg. Szükség van tehát két új változóra: *StartPoint* = a szakasz kezdőpontja (értékét az egérgomb lenyomásakor rögzítjük), valamint *OldPoint* = az előző rajzolt szakasz végpontja. Az egér minden elmozdításakor le kell törölnünk a régi szakaszt (*StartPoint* és *OldPoint* között), és újra kell rajzolnunk új „egéradta" pozíciójában (*StartPoint* és az egér *X,Y* között).

A két új változót (*StartPoint* és *OldPoint*) az űrlaposztályban fogjuk deklarálni privát mezőként.



6.9. ábra. Régi szakasz törlése, új szakasz rajzolása

Hogyan töröljünk le egy vonalat? Az nem jó, ha ráhúzunk egy fehér szakaszt, hiszen akkor végig fehér lesz, ott is ahol például egy másik (mondjuk piros) vonalat metsz. Valójában az eredeti háttérszínnek kellene valahogy visszaállnia. Erre alkalmazunk egy trükköt: rajzolás közben állítsuk be a toll rajzmódját (*Pen.Mode*) XOR vagy NotXOR-ra. Ha ezt tesszük, akkor rajzoláskor a toll- és a háttér színe között XOR vagy NotXOR műveletet hajt végre a rendszer. Ha pedig kétszer rajzolunk ugyanarra a helyre, akkor másodszori rajzolás után visszaáll az eredeti háttér. Ezt mutatja a 6.10. ábra.

Háttér színe	Toll színe	Első XOR	Második XOR	Első NotXOR	Második NotXOR
0	0	0	0	1	0
0	1	1	0	0	0
1	0	1	1	0	1
1	1	0	1	1	1

6.10. ábra. Rajzolás XOR és NotXOR módban

Vajon miért létezik XOR és NotXOR rajzmód is? A kettő közötti különbség az első rajzolás eredményében mutatkozik. Ha fekete a háttér (*Color=$000000*), akkor az XOR „átengedi" a toll színét, míg a NotXOR pont a toll „negatív" színét jeleníti meg. Ha viszont fehér háttérre rajzolunk (*Color=$FFFFFF*), akkor NotXOR rajzmódot használjunk.

Nézzük a kódot:

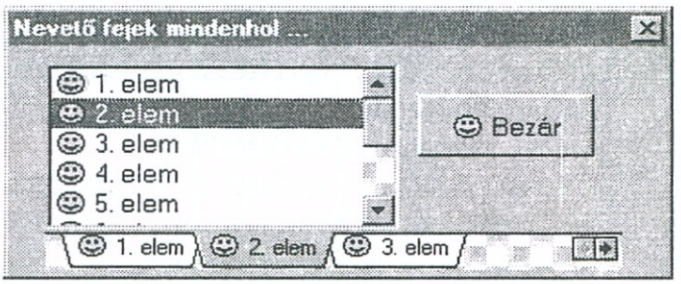
```
procedure TfrmRajzolo.KepMouseDown(Sender: TObject;
   Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
   if Button = mbLeft then {ha az egér bal gombját nyomták le}
      With Kep.Canvas Do
         begin
            InDraw:= True; {elkezdjük a rajzolást}
            case DrawingTool of
               dtLine: begin
                           StartPoint:= Point(X,Y);
                           OldPoint:= StartPoint;
                           Pen.Mode:= pmNotXOR;
                        end;
               dtFreeHand: Moveto(X,Y);
            end;
         end;
end;
```

```
procedure TfrmRajzolo.KepMouseMove(Sender: TObject;
                  Shift: TShiftState; X, Y: Integer);
begin
  If InDraw then {ennek köszönhetően nem történik rajzolás például
    akkor, amikor még az eszközsoron nyomjuk le az egérgombot, majd
    bevonszoljuk a kép fölé}
    With Kep.Canvas Do
      begin
        Case drawingTool Of
          dtLine: begin
                    {régi vonal letörlése}
                    Moveto(StartPoint.X, StartPoint.Y);
                    LineTo(OldPoint.X, OldPoint.Y);
                    {Új vonal kirajzolása}
                    Moveto(StartPoint.X, StartPoint.Y);
                    LineTo(X, Y);
                    {Régi pont elmentése}
                    OldPoint:= Point(X,Y);
                  end;
          dtFreehand: LineTo(X,Y);
        end;
      end;
end;

procedure TfrmRajzolo.KepMouseUp(Sender: TObject;
        Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  If Indraw then
    With Kep.Canvas Do
      begin
        InDraw:= False;
        case DrawingTool of
          dtLine: begin
                    {Vonal véglegesítése}
                    Pen.Mode:=pmCopy;
                    Moveto(StartPoint.X, StartPoint.Y);
                    LineTo(X, Y);
                  end;
          dtFreeHand: LineTo(X,Y);
        end;
      end;
end;
```

Fejlessze tovább a rajzolóprogramot! Lehessen ellipsziseket, téglalapokat, lekerekített sarkú téglalapokat is rajzolni, lehessen állítani a toll és az ecset jellemzőit. A kész alkalmazás megtalálható a DELPHI\DEMOS\DOC\GRAPHEX könyvtárban.

6.3.2 Grafika listadobozban és fülsorban

Készítsünk olyan listadobozt és fülsort, melynek elemei nemcsak szöveget, hanem grafikát is tartalmaznak. Az egyszerűség kedvéért minden elem előtt ugyanazt a „nevető fejet" jelenítsük meg. Sőt, a listában és fülsorban mindig azonos sorszámú elem legyen kijelölve: ha a listában kattintunk az 'i. elemre', akkor a fülsorban is az 'i. elem' feliratú fül kerüljön fókuszba, és fordítva: a fülekre való kattintáskor a listában a megfelelő elem jelölődjön ki.



6.11. ábra. Grafika a listadobozban és a fülsoron

Megoldás (🖫 6_OWNERD\OWNERDRAWP.DPR)

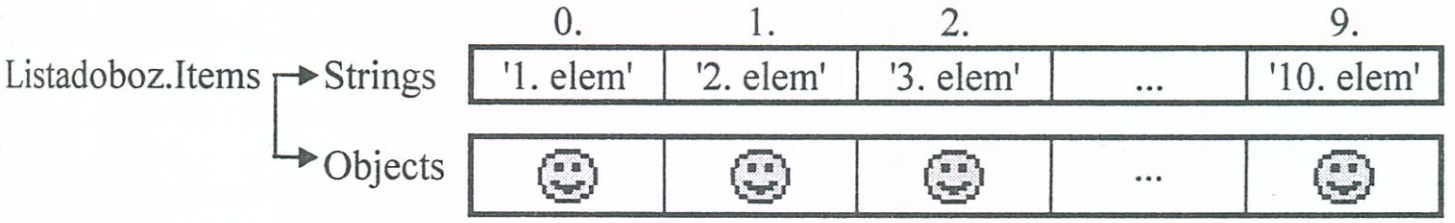
Bizonyos komponenseknél – mint a listadoboz, kombinált lista vagy a fülsor – lehetőség van különböző grafikák megjelenítésére. Windowsban sok helyen (például az Intézőben) találkozhatunk „rajzos" listadobozokkal, miért ne lehetne tehát Delphiből is ilyet létrehozni? Elég ezen komponensek *Style* jellemzőjét beállítanunk „saját rajzolásúra" (*OwnerDraw*). Ennek további változatai lehetnek: *OwnerDrawFixed* = minden elem egyforma méretű, vagy *OwnerDrawVariable* = az elemek különböző méretűek.
Ezzel saját magunkra vállaltuk az elemek kirajzolását, nekünk kell ezt majd kódból megvalósítanunk az *OnDrawItem* vagy *OnDrawTab* eseményekben (az első a listadoboznál és a kombinált listánál van, a második pedig a fülsornál). Sőt, ha az elemek még különböző magasságúak is, akkor az egyes elemek méretét is nekünk kell kódból kiszámolnunk (az *OnMeasureItem* vagy *OnMeasureTab* eseményekben).

Feladatunkban tehát fix magasságú saját rajzolású listadobozt és fülsort fogunk kezelni. Tervezzük meg űrlapunkat (*frmOwnerDraw*)!

Komponens	Jellemző	Érték
Listadoboz: TListBox	Style	lbOwnerDrawFixed
Fulsor: TTabSet[1]	Style	tsOwnerDraw
btnBezar: TBitBtn	Caption	'Bezár'
	Glyph	'SMILE.BMP'

[1] A Win32 palettán található fülsoron (*TTabControl*) nem lehet grafikát megjeleníteni. Ezúttal az 1-es Delphi-ben bevezetett *TTabset* komponenst kell használnunk (Win3.1 paletta).

A listadoboz és a fülsor szinkronizált működése csak akkor képzelhető el, ha azonos számú és megegyező nevű elemeket tartalmaznak. Így hát, ahelyett, hogy tervezéskor minden elemet két példányban adnánk meg (egyet a *Listadoboz*nál, másikat a *Fülsor*nál), a listadoboz és fülsor feltöltését futási időre halasztjuk.
Az elemeiket tartalmazó jellemzők (*Items* és *Tabs*) *TStrings* típusú objektumok. A *TStrings* osztályban nemcsak karakterláncokat, hanem egyéb információkat is lehet tárolni (lásd 4. fejezet, 4.14.2. feladat): a karakterláncokat a *Strings[I:Integer]:String* tömbben, míg az „egyebet" az *Objects[I:Integer]:TObject* tömbben. És mivel az „egyéb" *TObject* típusúnak lett deklarálva, a *TObject* pedig az összes Delphibeli osztály közös őse, ez azt jelenti, hogy akármilyen objektumot el lehet ebben a tömbben helyezni. Mi most képeket (*TBitmap*) rakunk bele (lásd 6.12. ábra). Így elvileg minden listaelem előtt más és más képet jeleníthetnénk meg (lehetnének szomorú fejek is közöttük).



6.12. ábra. A listadoboz elemei szöveget és képet tartalmaznak

Nézzük a listadoboz és fülsor feltöltésének kódját:

```
procedure TfrmOwnerDraw.FormCreate(Sender: TObject);
Var I:Integer;
    Bitmap: TBitmap;
begin
  Bitmap:= TBitmap.Create;
  Bitmap.LoadFromFile('SMILE.BMP');
  For I:=1 To 10 Do
    Begin
      Listadoboz.Items.AddObject(IntToStr(I)+'. elem', Bitmap);
      Fulsor.Tabs.AddObject(IntToStr(I)+'. elem', Bitmap);
    End;
  Fulsor.TabIndex:=0;
end;
```

Következhet a rajzolás! A listadoboz *OwnerDrawFixed* stílusú, így a rajzolást az *OnDrawItem* eseményében kell megvalósítanunk. Ez külön-külön minden egyes kirajzolandó listaelemre meghívódik, tehát egyszerre csak egy listaelemet kell benne megrajzolnunk: a paraméterben megadott indexűt (*Index*), a paraméterként megadott téglalapnyi területben (*Rect*).

6.13. ábra. Mit, hova rajzoljunk?

Ez maga a listadoboz

```
procedure TfrmOwnerDraw.ListadobozDrawItem(Control: TWinControl;
        Index: Integer; Rect: TRect; State: TOwnerDrawState);
var
   Bitmap: TBitmap;
begin
   With (Control as TListBox).Canvas Do
      begin
         FillRect(Rect); {letöröljük a kirajzolandó elem területét}
         {kirajzoljuk az elemhez tartozó képet}
         Bitmap := TBitmap((Control As TListBox).Items.Objects[Index]);
         If Bitmap <> nil Then
            begin
               {Draw(Rect.Left + 2, Rect.Top,Bitmap);}
               BrushCopy(Bounds(Rect.Left + 2, Rect.Top, Bitmap.Width,
                  Bitmap.Height), Bitmap,
                  Bounds(0, 0, Bitmap.Width, Bitmap.Height), clWhite);
            end;
         {megjelenítjük a szöveget is}
         TextOut(Rect.Left + Bitmap.Width + 4, Rect.Top,
              (Control As TListBox).Items[Index])
      end;
end;
```

Van itt egy érdekesség: ha a képet a *Draw* metódussal rajzolnánk ki, akkor a **kijelölt** elem képe is megmaradna fehér hátterűnek. Próbálja ki! Mi azt szeretnénk, ha ilyenkor a kép háttere is „bezöldülne", akárcsak a kijelölt elemé. Ezért használjuk a *BrushCopy* metódust. Szintaxisa:

```
BrushCopy( Hova:TRect;           MelyikBitterkepet: TBitmap;
           Melyikrészét:TRect;   HelyettesítendőSzín:TColor);
```

Ez egy bittérkép adott részét rajzolja ki egy céltéglalapnyi területre úgy, hogy a bittérképben az egyik színt helyettesíti a célterület háttérszínével. A 'SMILE.BMP'-nek fehér a háttere, így most ezt kell felcserélnünk a háttérszínnel.

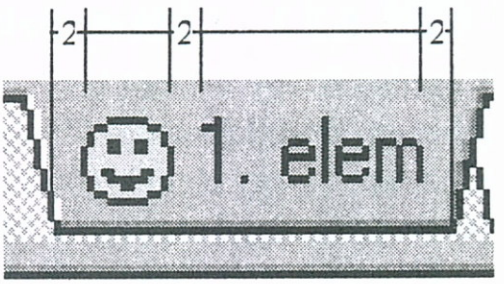
Használtuk a *Bounds* függvényt is. Ez visszaadja a paraméterként megadott bal-felső sarkú, adott szélességű és magasságú téglalapot. Ugyanezt teszi a *Rect* függvény is, csak annak paraméterként a terület bal-felső és jobb-alsó sarkainak koordinátáit kell megadnunk. A két függvény közül mindig a kényelmesebben alkalmazhatót kell használnunk. Esetünkben két érv is szól a *Rect* ellen:

- egyrészt most a terület szélességét és magasságát ismerjük, a jobb-alsó sarok koordinátáit ebből kellene kiszámolni;
- másrészt, eljárásunkban már van egy *Rect* azonosító (a paramétert jelöli), így a *Rect* függvényt csak minősítéssel érhetjük el (*Classes.Rect(...)*)

Rajzoljunk a fülsorra is. Ennek stílusát *tsOwnerDraw*-ra állítjuk be. A fülsornál két eseményt kell lekódolnunk: az *OnMeasureTab*-ban ki kell számolnunk a kirajzolandó fül szélességét, az *OnDrawTab*-ba pedig a rajzolást kell beépítenünk.

Ebben a változó paraméterben kell kiszámolnunk az Index-edik fül szélességét

```
procedure TfrmOwnerDraw.FulsorMeasureTab(Sender: TObject;
                          Index: Integer; var TabWidth: Integer);
var
  Bitmap: TBitmap;
begin
  Bitmap:=TBitmap(Fulsor.Tabs.Objects[Index]);
  if Bitmap <> nil then
    With Fulsor.Canvas Do
      TabWidth:= 2+Bitmap.Width + 2 +
      TextWidth(Fulsor.Tabs[Index])+2;
end;
```



6.14. ábra. Mit, hova rajzoljunk?

Erre a vászonra kell rajzolni

```
procedure TfrmOwnerDraw.FulsorDrawTab(Sender: TObject;
    TabCanvas: TCanvas; R: TRect; Index: Integer; Selected: Boolean);
var
  Bitmap: TBitmap;
begin
  With TabCanvas Do
    begin
      FillRect(R);
      Bitmap := TBitmap(Fulsor.Tabs.Objects[Index]);
      if Bitmap <> nil then
        begin
          BrushCopy(Bounds(R.Left + 2, R.Top+2, Bitmap.Width,
            Bitmap.Height), Bitmap,
            Bounds(0, 0, Bitmap.Width, Bitmap.Height), clWhite);
        end;
      TextOut(R.Left + Bitmap.Width + 4, R.Top+2,Fulsor.Tabs[Index])
    end;
end;
```

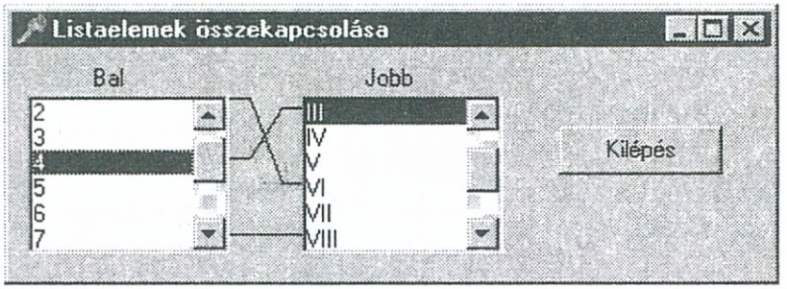
A listadoboz és a fülsor szinkronizálása a következő két metódussal valósul meg:

```
procedure TfrmOwnerDraw.ListadobozClick(Sender: TObject);
begin
   Fulsor.Tabindex:=Listadoboz.Itemindex;
end;

procedure TfrmOwnerDraw.FulsorClick(Sender: TObject);
begin
   Listadoboz.Itemindex:=Fulsor.Tabindex;
end;
```

Nézze át a DELPHI\DEMOS\OWNERLST\FONTDRAW.DPR mintapéldát! Egy változó magasságú szövegeket tartalmazó listadobozt talál majd benne.

6.3.3 Listaelemek grafikai összekötése (a *TList* osztály használata)

Készítsünk egy űrlapot két listadobozzal. A bal és jobb listadobozok elemeit vonszolással lehessen összekötni úgy, hogy görgetéskor a listaelemekhez tartozó vonalak kövessék a mozgást; ha egy elem kigördül a listából, akkor vonala ragadjon a doboz tetejéhez vagy aljához, attól függően, hogy a listaelem fölül vagy alul tűnt-e el.



6.15. ábra. Összeköthető listadobozok

Megoldás (🖫 6_LISTAK\PLISTAK.DPR)

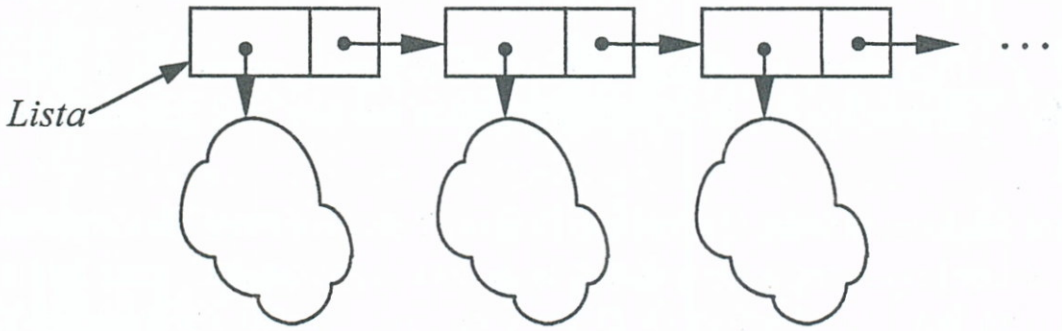
Tervezzük meg az űrlapot:

Komponens	Jellemző	Érték
lbBal: TListBox	Items	1, 2...10
lbJobb: TListbox	Items	I, II...X
btnKilepes: TButton	Caption	'Kilépés'

Az egérvonszolással tulajdonképpen összekapcsolunk a bal lista és a jobb lista egy-egy elemét. A kapcsolat jelzésére megjelenik egy vonal. Ezt a vonalat újra kell tudnunk rajzolni később is, a listadoboz (-ok) görgetése után. Ennek érdekében a létrehozott kapcsolatokat tárolni fogjuk egy *TList* típusú listára felfűzve.

6.3.3.1 TList osztály

- Szerepe: általános konténer osztály. Elemek tárolására, visszakeresésére, rendezésére alkalmas. Bármit el lehet benne helyezni: az egész számoktól a választógombokig. Tulajdonképpen ez egy mutatókat tartalmazó lista, melynek elemei (mutatói) bármire mutathatnak: objektumokra és nem objektumokra is. Elemeit indexeléssel érhetjük el.



6.16. ábra. A *Lista:TList* szerkezete. A „felhőcskékben" akármi lehet!

- Fontosabb jellemzői:
  - **Items [Index:Integer] : Pointer**
    Ez egy tömb típusú jellemző, melynek segítségével a listaelemekre indexeléssel hivatkozhatunk.
  - **Count: Integer**
    A listában jelenleg létező elemek száma. Első elem indexe = 0, utolsóé pedig *Count*-1.
  - **Capacity:Integer**
    Ez a jellemző a lista kapacitását mutatja, vagyis azt, hogy hány elem fér el maximálisan a listában további helyfoglalások nélkül. Természetesen akkor sincs baj, ha egy megtelt listában (*Count* = *Capacity*) akarunk egy új elemet elhelyezni; ekkor a „jövevény" számára automatikusan lefoglalódik hely. Az apránkénti helyfoglalások viszont több időt vesznek igénybe, mint ha az egészet egyszerre, egy lépésben tennénk, ezért, ha előre ismerjük a lista elemeinek számát, akkor ajánlatos ezt az értéket már előre beállítani a *Capacity*-be.
    A megtelt lista tehát nem okoz problémát. Az is előfordulhat viszont, hogy maga a virtuális memória telik meg, erre egy *EOutOfMemory* kivétel figyelmeztet. (A virtuális memória mérete főképpen az operatív tár- és a háttérállomány méreteitől függ.)
- Fontosabb metódusai:
  - **Create**: lista inicializálása.
  - **Add (Elem:Pointer)**
    Új elem elhelyezése a lista végén.

- **Insert (Index:Integer; Elem:Pointer)**
  Új elem beszúrása a lista *Index*-edik pozíciójába.
- **Delete (Index:Integer)**
  Az *Index*-edik elem törlése a listából.
- **Destroy**: A lista destruktora. Ez maga után vonja a lista főgerincének (a mutatók által lefoglalt memóriaterület) felszabadítását (lásd 6.16. ábra).

Figyelem, a „felhőcskék" által lefoglalt memóriát a destruktor nem szabadítja fel, erről szükség esetén nekünk kell gondoskodnunk!

A lista felszabadítására ajánlatos a *Destroy* helyett a ***Free*** metódust használni; ez csak akkor hívja meg a destruktort, ha a lista „egészséges", azaz inicializálták, de még nem lett felszabadítva (a *Destroy* és *Free* közötti különbségekről már a 3. fejezetben beszéltünk).

Térjünk vissza listadobozos feladatunkhoz! A listán most kapcsolatokat (bal listaelem, jobb listaelem) szeretnénk elhelyezni. Definiáljuk a kapcsolat típusát (*TKapcs*), majd deklaráljuk a kapcsolatokat tartalmazó listát:

```
type
   PKapcs = ^ TKapcs;
   TKapcs = Record
      BalIndex,              {a bal lista melyik elemét kötöttük össze}
      JobbIndex:Integer;     {a jobb lista melyik elemével}
   End;

   TfrmListak = class(TForm)
         ...
      private
         KapcsokLista:TList;
         ...
   End;
```

A lista inicializálását a program elejére, felszabadítását pedig a legvégére kell beírnunk:

```
procedure TfrmListak.FormCreate(Sender: TObject);
begin
  {A TList.Create-nek nincsenek paraméterei}
  Kapcsoklista:= TList.Create;
end;

procedure TfrmListak.FormDestroy(Sender: TObject);
var Kapcs:PKapcs;
begin
   While KapcsokLista.Count > 0 Do {előbb leürítjük a listát}
      begin
         Kapcs:=KapcsokLista[0];
```

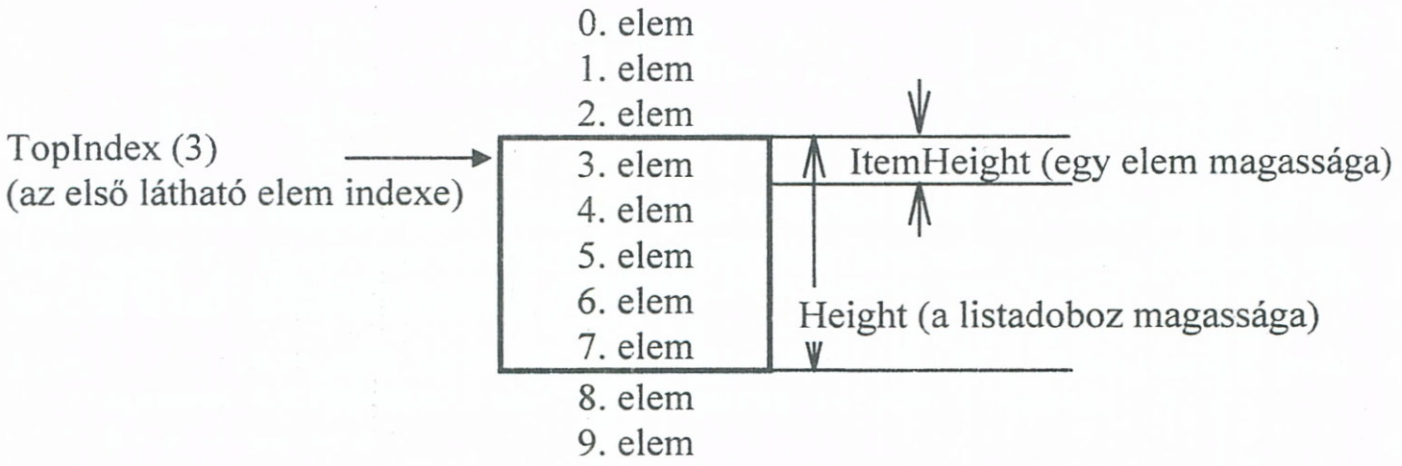
```
        Dispose(Kapcs);
        KapcsokLista.Delete(0);
      end;
  KapcsokLista.Free;
end;
```

Következhet a vonszolás (a *drag&drop* technikáról már a 4. fejezetben beszéltünk, ezért most nem részletezzük):

```
{Mindkét lista OnMouseDown eseményjellemzőjét erre a metódusra irá-
nyítjuk}
procedure TfrmListak.ListakMouseDown(Sender: TObject;
        Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  If (Button = mbLeft) And (Sender is TListBox) then
    With Sender As TListBox Do
      If ItemAtPos(Point(X,Y), True) >=0 then
        BeginDrag(False);
end;

{Mindkét lista OnDragOver eseményjellemzőjét erre a metódusra irányít-
juk.}
procedure TfrmListak.ListakDragOver(Sender, Source: TObject;
          X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  If (Source is TListBox) And (Source <> Sender) Then
    Accept:=True;
end;

{Mindkét lista OnDragDrop eseményét erre a metódusra irányítjuk.}
procedure TfrmListak.ListakDragDrop(Sender, Source: TObject;
          X, Y: Integer);
var
  Kapcs:PKapcs;
begin
  Kapcs:= New(PKapcs); {létrehozunk egy új kapcsolatot}
  {a BalIndex mezőbe a bal listaelem indexét, a JobbIndex-ba pedig a
  jobbét helyezzünk majd el.}
  If Source = lbBal Then {ha balról jobbra vonszoltunk egy elemet}
    Begin
      Kapcs^.BalIndex:= (Source as TListBox).ItemIndex;
      Kapcs^.JobbIndex:= (Sender as TListBox).
                    ItemAtPos(Point(X,Y),True);
    End
  Else{ha jobbról balra vonszoltunk egy elemet}
    Begin
      Kapcs^.BalIndex:= (Sender as TListBox).
                        ItemAtPos(Point(X,Y),True);
      Kapcs^.JobbIndex:= (Source as TListBox).ItemIndex;
```

```
    End;
  {az új kapcsolatot felfűzzük a listára}
  Kapcsoklista.Add(Kapcs);
  {kirajzoljuk a vonalat}
  KapcsRajzol(Kapcs^.BalIndex, Kapcs^.JobbIndex);
end;
```

Egy kapcsolatot jelző vonal kirajzolására bevezetjük a *KapcsRajzol* privát metódust. Ennek két paramétere van: a bal lista melyik elemét (*BalIndex*) kösse össze a jobb lista melyik elemével (*JobbIndex*).
A rajzolásnál figyelnünk kell arra, hogy a vonal ne hagyhassa el a listadoboz felső és alsó peremét. Ehhez nyújt segítséget a 6.17. ábra.



6.17. ábra. Egy listadoboz melynek elemei „nem férnek a bőrükbe”

A rajzolás az űrlap vásznára történik, így a koordinátákat az űrlap bal-felső sarkától kell mérnünk.

```
procedure TfrmListak.KapcsRajzol(BalIndex, JobbIndex:Integer);
var x1, y1, x2, y2:Integer;
begin
  With lbBal Do {x1,y1 kiszámolása}
    begin
      x1:= Left+Width;
      If BalIndex < TopIndex Then {ha kicsorogna felül}
        y1:= Top + 1
      else if BalIndex < TopIndex + Height div ItemHeight Then
        {ha a listadoboz látható elemei közé esik}
        y1:= Top + (BalIndex-TopIndex)*ItemHeight + ItemHeight div 2
      else {ha kicsorogna alul}
        y1:= Top + Height - 1;
    end;
  With lbJobb Do {x2,y2 kiszámolása}
    begin
```

```
        x2:= Left;
        If JobbIndex < TopIndex Then {ha kicsorogna felül}
           y2:= Top+1
        else if JobbIndex < TopIndex + Height div ItemHeight Then
           {ha a listadoboz látható elemei közé esik}
           y2:=Top +(JobbIndex-TopIndex)*ItemHeight + ItemHeight div 2
        else {ha kicsorogna alul}
           y2:= Top + Height-1;
     end;
  With Canvas Do {rajzolás}
     begin
        Polyline([Point(x1,y1), Point(x1+10,y1),
                  Point(x2-10,y2), Point(x2,y2)]);
     end;
end;
```

Futtassa le így az alkalmazást! A vonalak megjelennek, de ott is maradnak, hiába görgetjük a listadobozokat.

A vonalakat újra kellene rajzolnunk minden görgetés után. Igen ám, de a *TListbox*-nak nincs *OnScroll* eseményjellemzője! Mi legyen? A 15. fejezetben majd készítünk *OnScroll* jellemzővel rendelkező listadobozt (15.7. pont), de addig is oldjuk meg valahogy a problémát. A listadoboz *OnClick* eseménye itt nem használható, hiszen ez nemcsak a gördítősávra való kattintáskor következik be, hanem akkor is, amikor egy listaelemre kattintunk; ráadásul a listadoboz billentyűzetről is görgethető (a nyílbillentyűkkel), a vonalaknak olyankor is követniük kellene a mozgást.
A megoldás a következő: állítsuk be a listadobozok stílusát *Style = lsOwnerDrawFixed*-re. Amikor görgetjük a listát és egy friss listaelem megjelenik, egészen biztosan meghívódik az *OnDrawItem* esemény; ekkor fogjuk a vonalakat újrarajzolni.

```
procedure TfrmListak.ListakDrawItem(Control: TWinControl;
        Index: Integer; Rect: TRect; State: TOwnerDrawState);
var i:Integer;
begin
  {kiírjuk az Index-edik listaelemet}
  With Control as TListbox Do
     begin
        Canvas.Fillrect(Rect);
        Canvas.TextOut(Rect.Left,Rect.Top,Items[Index]);
     end;
  {letöröljük a két lista közötti űrlapterületet,
   majd újrarajzoljuk a kapcsolatokat a megfelelő helyen.}
  With Canvas Do {ez már az űrlap vászna}
     begin
        Brush.Color:= self.Brush.Color;
        Fillrect(Bounds(lbBal.Left+lbBal.Width,lbBal.Top,lbJobb.Left,
                        lbJobb.Top+lbJobb.Height));
```

```
        For i:=0 to Kapcsoklista.Count-1 Do
          KapcsRajzol( PKapcs(Kapcsoklista[i])^.BalIndex,
                       PKapcs(Kapcsoklista[i])^.JobbIndex);
    end;
end;
```

A *ListakDrawItem* metódus mindkét listadoboznál használható, hiszen a *Control* paramétere mindig a kirajzolandó listadoboz mutatóját tartalmazza.

Futtassa le az alkalmazást! Most már (remélhetőleg) minden jól működik.

6.4 Nyomtatás

Delphi alkalmazásainkból nyomtathatunk szövegeket és ábrákat a *Printer* objektum segítségével (6.4.1. pont). Ezen kívül – a csak szöveges adatok nyomtatására – van egy másik, kényelmesebb megoldás is (6.4.2. pont). Ha viszont egy adatbázis bizonyos adatait szeretnénk kinyomtatni, akkor arra már a speciális *QuickReport* komponenscsaládot használjuk (pusztán kényelmi okokból), erről bővebben majd a 13. fejezetben lesz szó.

6.4.1 Nyomtatás a Printer objektummal

A *Printer:TPrinter* objektum deklarációja és inicializálása a *Printers* egységben található, ezért használatakor ezt az egységet alkalmazásunkba be kell építenünk.

TPrinter osztály

- Szerepe: összegyűjtve tartalmazza a nyomtatással kapcsolatos információkat. Rajzvászna segítségével nyomtathatunk a kiválasztott nyomtatóra.
- Fontosabb jellemzői:
  - **Canvas:TCanvas**
    A nyomtató rajzvászna. Mindaz, amit erre rajzolunk, a nyomtatóra kerül.
  - **Printing:Boolean**
    Jellemző, melynek igaz értéke azt jelzi, hogy a nyomtatás folyamatban van.
  - **PageNumber:Integer**
    A pillanatnyilag nyomtatás alatt álló oldal számát tartalmazza.
  - **PageHeight, PageWidth: Integer**
    A nyomtatandó lap méretei pixelekben.
  - **Copies: Integer**
    A kinyomtatandó példányszám.

A nyomtatással kapcsolatos adatokat csakis a nyomtatóbeállítási párbeszédablakok (*PrintDialog* és *PrinterSetupDialog*) segítségével állítsuk be. A *Printer* jellemzőit (a *Canvas* kivételével) az adatok programból történő lekérdezésére használjuk.

- Fontosabb metódusai:
  - **BeginDoc**: a nyomtatás elkezdésekor kell meghívnunk. A nyomtatási előkészületeket végzi el.
  - **EndDoc**: a kinyomtatandó szövegek vagy ábrák nyomtatóra való sikeres kiküldése után hívjuk meg. Hatására megkezdődik a tényleges nyomtatás. Ha valamilyen okból kifolyólag szövegeinket vagy ábráinkat nem tudtuk kiküldeni a nyomtatóhoz, akkor az *Abort* metódust hívjuk az *EndDoc* helyett.
  - **Abort**: hatására az aktuális nyomtatandó dokumentum törlődik a sorból.
  - **NewPage**: lapdobást, új lap elkezdését idézi elő.

A **szövegek, ábrák kinyomtatása** a *Printer.Canvas* metódusaival történik a következő lépések betartásával:

1. Nyomtató-beállítások (esetleg): erre használhatjuk a következő két párbeszédablakot (megjelenítésük az *Execute* metódusukkal valósul meg):
   - *PrintDialog*: nyomtatandó tartomány, példányszám stb. beállítása
   - *PrinterSetupDialog*: papírméret, tájolás, adagolás stb. beállítása

   A párbeszédablakokban beállítottak automatikusan érvényesülnek majd nyomtatáskor, ezzel nem kell a kódban külön törődnünk.
2. Nyomtatás elkezdése: *Printer.BeginDoc*
3. Írás, rajzolás a nyomtatóra: *Printer.Canvas.TextOut(...), Printer.Canvas.Draw(...)...*
4. Nyomtatás befejezése: *Printer.EndDoc*

6.4.2 Szövegek nyomtatása

Ha csak szöveget szeretnénk kinyomtatni, akkor erre használhatunk egy másik nyomtatási technikát is:

1. Létrehozunk egy szöveges állományváltozót, majd hozzárendeljük a nyomtatóhoz (A *Printers* egységben található *AssignPrn* eljárással).
2. Kiírjuk a nyomtatandó szöveget a szöveges állományba.

Ez a megoldás kényelmesebb az előbbinél, mivel nem kell minden sornál külön megadnunk a koordinátákat (*TextOut (X,Y:Integer; Szöveg:String)*).

6.4.3 Grafikus ↔ karakteres nyomtatás

Biztosan észrevette már a kedves Olvasó, hogy a Windows rendszerekben a nyomtatás grafikusan történik. Ez persze jó, mert így ábrákat is tudunk nyomtatni, és számtalan betű-

típust használhatunk. A nyomtatás viszont emiatt lassúbb. A lassúságot főleg a mátrix-nyomtatóknál vesszük észre: ezen egy sor szöveg grafikus kinyomtatásáért a nyomtatófej általában többször is végigszalad a sor felett. Vannak esetek, programok, melyekben kevésbé igényes listákat szeretnénk **gyorsan** produkálni mátrixnyomtatóra úgy, ahogyan a hagyományos DOS-os világban megszoktuk.

Nézzük, hogyan lehet karakteresen nyomtatni Windowsból!

Az előző pontokban bemutatott nyomtatási módszerekkel alapértelmezés szerint grafikus nyomtatást érünk el. Ha azt akarjuk, hogy karakteresen, azaz gyorsan jelenjenek meg a nyomtatón szövegeink, akkor ennek legegyszerűbb módja egy speciális nyomtató-driver használata. Ez az ún. ***„Generic/Text Only"*** meghajtó, mely csak szöveget képes nyomtatni, karakteres módban, formátum nélkül.
E lehetőség használatának előfeltétele, hogy ez a meghajtó telepítve legyen gépünkön (*Printers/Add* printer a *Start* menüben). A nyomtató kezelése ugyanúgy zajlik, mint bármelyik más nyomtatóé (ugyanazokkal a párbeszéd-ablakokkal).

Egyszerűnek tűnik minden, mégis föl kell hívnunk a figyelmet valamire:

A mátrix-nyomtatók karakteres üzemmódja nagyon hasonlít a karakter-alapú felhasználói felülettel rendelkező programok videó-üzemmódjaihoz (például a DOS). Ezekben a szokásos alfanumerikus karaktereken kívül olyanokra is szükség van, amelyekkel vonalakat, táblázatokat lehet előállítani a monitoron, illetve a nyomtatón.
A grafikus felületek esetén (például Windows) az ilyen „ínyencségeket" könnyebb közvetlenül – vonalként, táblázatként – pontokból megjeleníteni, mint „vonal-szerű" karakterekből. Emiatt a Windowsban használt karakterkészlet eltér a DOS-ban használatostól. Mind a kettő 256 karaktert tartalmaz, melynek első 128 karaktere azonos (ASCII karakter-készlet). A felső 128 karakternél vannak a különbségek: a Windowsban csak sajátos nemzeti karakterek és szimbólumok kaptak helyet (ANSI szabvány, magyar nyelv esetén 1250 kódlap), míg a DOS esetén itt találhatjuk a „rajz-szimuláló" karaktereket is (OEM, magyar nyelv esetén CP 852). Így fordul elő, hogy azonos karaktereknek a két rendszerben más-más kód felel meg. Vagyis, ha Windows-ból egyenesen kiküldjük a karaktereket a nyomtatóra, akkor a magyar nyelv sajátos betűi hibásan jelennek majd meg, mivel az egyes karakter-kódok értelmezése a nyomtatón múlik, pontosabban a bele épített „karakter-generátoron". A nyomtató DOS-osan értelmezi az általunk windowsosan küldött kódokat. Így például az ö-ből ÷ lesz.
A megoldást a különböző kódlapok közötti konverzió jelenti (az ANSI 'ö'-ből OEM 'ö' lesz, és ez lesz kinyomatva). Emiatt fontos az, hogy a *Generic/Text Only* tulajdonságaiban megfelelően állítsuk be az a kódlapot, mely a magyar nyelv sajátos karaktereit minél jobban leképezi (CP 850 – ideális volna a 852, de ilyennel nem rendelkezem). Semmi esetre se hagyjuk meg az alapértelmezett *Driver Default* beállítást, mert ez konverzió nélkül nyomtat!

Egy ily módon beállított karakter-alapú nyomtató ugyanúgy működik hálózatos környezetben, mint a szokásos grafikus nyomtatók (kiajánlás, hálózati nyomtatás...).

A karakteres nyomtatásnak van egy másik módja is, mely ugyan bonyolultabb kissé, de előnyösebb lehet az eleve **DOS-os kódlappal megírt szövegek esetén**. A létrehozott állományt nyomtassuk ki az MS-DOS-ból ismert *Print* paranccsal (*Print SZOVEG.TXT*). Egy Delphi alkalmazásból ezt a *WinExec* API függvénnyel hívhatjuk meg, mely egy alkalmazás futtatására alkalmas.
Például:

```
WinExec (StrNew('Print SZOVEG.TXT'), SW_HIDE);
```

Hogyan jelenjen meg az alkalmazás ablaka

Az elindítandó alkalmazás neve és paraméterei C típusú karakterláncban

Ha több helyi nyomtató között válogathatunk, vagy ha nyomtatónk a hálózat egy másik gépére van csatlakoztatva, akkor használjuk a *Print* parancs '*/d*' kapcsolóját:

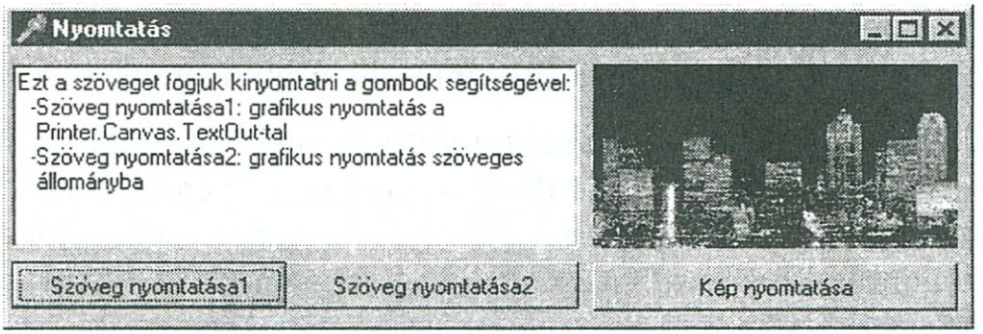
```
{Nyomtatás a helyi gép LPT2-re csatlakoztatott nyomtatójára}
Print /d:LPT2 SZOVEG.TXT

{Nyomtatás a hálózat Pentike gépére csatlakoztatott nyomtatóra. A
nyomtató kiajánlási neve: Citi}
Print /d:\\Pentike\Citi SZOVEG.TXT
```

Figyelem! Itt semmiféle konverzió nem történik, emiatt csak az eleve DOS-os szövegek esetén működik helyesen. Igaz, nem DOS-os szövegek esetén programból előírhatunk konverziót is (az *AnsiToOem* vagy Delphi 3-ban *CharToOem* függvények segítségével), de mivel a PRINT-nek egy állományt kell paraméterként megadni, emiatt a konvertált szöveget le is kellene menteni egy újabb állományba, majd a nyomtatás befejeztével, ezt le is kell törölnünk a programból. Nem DOS-os szövegek kinyomtatására az első megoldás (*Generic Text/Only*) sokkal kényelmesebb.

Mindezen nyomtatási technikák begyakorlására oldjuk meg a következő feladatot:

6.4.4 Feladat: szövegek és ábrák, karakteres és grafikus nyomtatása

Űrlapunkon helyezzünk el egy többsoros szerkesztődobozt és egy képet. Nyomtassuk ki ezek tartalmát. A szöveg nyomtatásánál próbáljuk ki mindkét bemutatott módszert.



6.18. ábra. Szöveg és grafika kinyomtatása

Megoldás (🖫 6_NYOMT\PNYOMTAT.DPR)

Helyezzük el az űrlapon a *Szoveg:TMemo* és a *Kep:TImage* komponenseket, valamint a három nyomtatási gombot (6.18. ábra). Nézzük a gombok kattintására írt kódot:

```
procedure TfrmNyomtatas.KepNyomtatasClick(Sender: TObject);
begin
   If PrintDialog.Execute then
      With Printer Do
         begin
            BeginDoc;
            Canvas.Draw(Canvas.Cliprect.Left, Canvas.Cliprect.Top,
                        Kep.Picture.Graphic);
            EndDoc;
         end;
end;

procedure TfrmNyomtatas.szovegNyomtatas1Click(Sender: TObject);
Var i:Integer;
begin
   With Printer, Canvas Do
      begin
         BeginDoc;
         Font:= Szoveg.Font;
         For i:=0 to Szoveg.Lines.Count-1 Do
            TextOut(Cliprect.Left,ClipRect.Top+i*TextHeight('W'),
                    Szoveg.Lines[i]);
         EndDoc;
      end;
end;

procedure TfrmNyomtatas.SzovegNyomtatas2Click(Sender: TObject);
var   PrinterFile:System.Text;
      i:Integer;
begin
   AssignPrn(PrinterFile);
   Rewrite(PrinterFile);
   For i:=0 to Szoveg.Lines.Count-1 Do
```

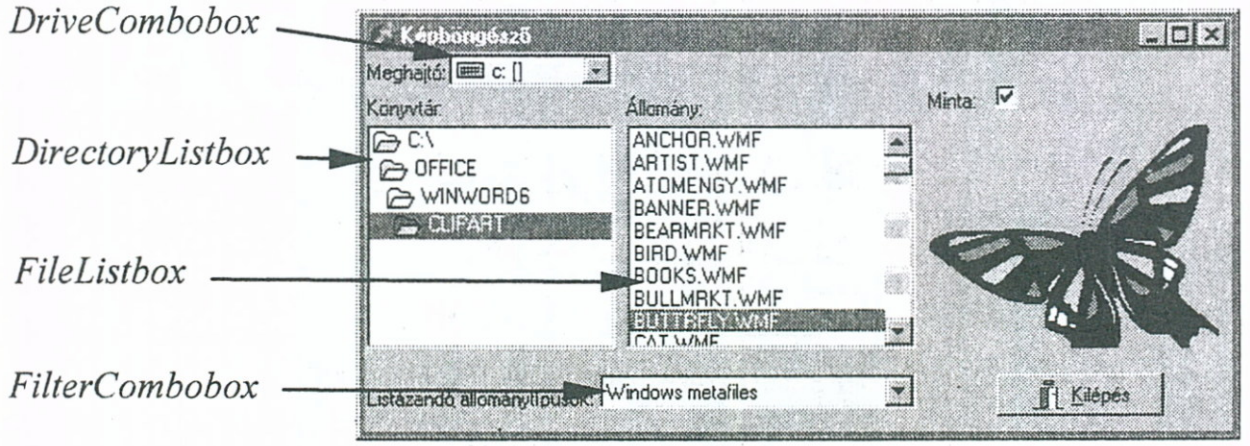
```
    WriteLn(PrinterFile, Szoveg.Lines[i]);
  CloseFile(PrinterFile);
end;
```

Tesztelje le az alkalmazást!

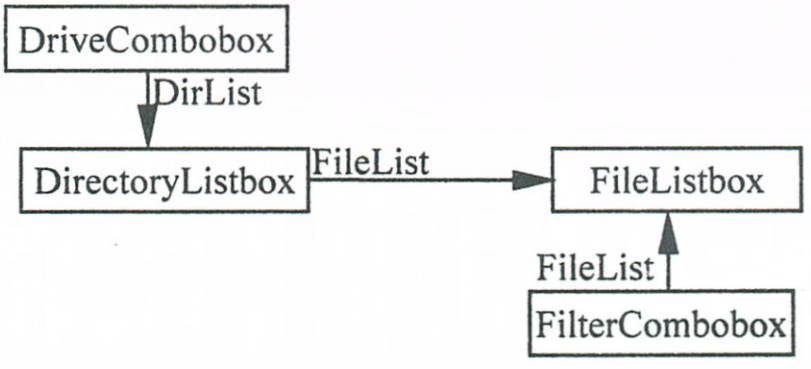
Feladatok

Képböngésző (6_BONGESZO\PBONGESZ.DPR)

Készítsen egy képböngésző párbeszédablakot a grafikus állományok tartalmának megtekintésére. A párbeszédablakban lehessen meghajtót és azon belül könyvtárat váltani, valamint állíthassuk be a listázandó állománytípust is. Ha a *Minta* jelölőnégyzet ki van pipálva, akkor a kiválasztott állomány tartalmát jelentse meg a mintaképen; egyébként a mintakép legyen üres.



Tipp: A megoldáshoz használja a Win3.1 palettán található és az ábrán is feltüntetett komponenseket (*TDriveCombobox*...). Együttműködésük érdekében kapcsolja ezeket össze a következőképpen:



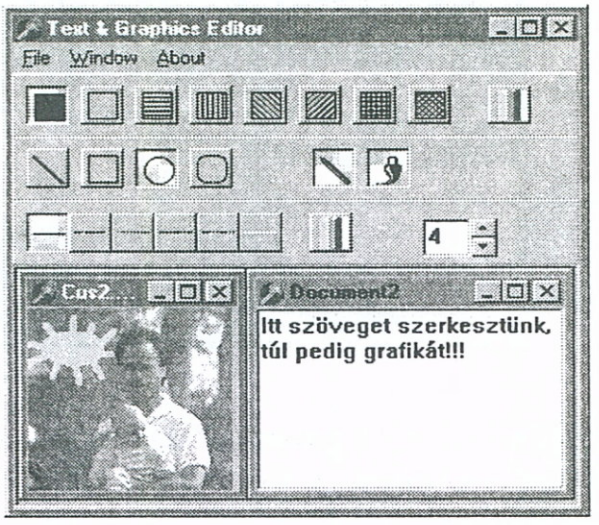
Dinamikus üdvözlőlap

Készítsen barátainak egy dinamikus karácsonyi üdvözlőlapot! Lehet rajta pezsgőspohár (amiben természetesen pezseg az ital), körülötte jókívánságokkal, vagy ábrázolhat egy havas tájat (persze dinamikus hóeséssel)... Az üdvözlőlapot lehessen kinyomtatni!

Tipp: Az ábra statikus részeit rajzolja meg egy rajzolóprogram segítségével (például Paintbrush), majd mentse el állományba. Delphiből ezt a képet jelenítse meg egy *TImage* komponensen, majd programból (például egy *TTimer* eseményében) rajzolja rá a dinamikus elemeket.

MDI típusú kép- és szövegszerkesztő

Egészítse ki az előző fejezetben írt szövegszerkesztő alkalmazást! Ne csak szöveget, hanem grafikát (BMP) is lehessen benne szerkeszteni, betölteni, ill. lementeni, természetesen külön ablakban! Legyenek tehát szöveges ablakaink és képablakaink. Továbbá, a szöveget és képeket ki is lehessen nyomtatni!



II. rész
Adatbázisok

E rész tanulmányozásához szükséges előismeretek

Az adatbázisok kezeléséről beszélni egy bonyolult feladat főleg, ha mindezt a Delphi kapcsán tesszük. Bonyolult azért, mert mindaddig nem beszélhetünk adatbázis-kezelésről egy konkrét környezetben, ameddig nem vagyunk tisztában az adatbázisok fogalmával, tervezésével, lekérdező nyelvével – az SQL-lel. Ugyanakkor, ha a jelen könyvben ezekről is szó esne, akkor helyszűke miatt, pont a könyvünk szempontjából lényegnek tekintett Delphi megvalósítási technikák bemutatása maradna el. És mivel a Delphiben lehetőségeink korlátlanok, nagyon sok mindenről kell beszélnünk.
E „vívódás" eredményeképpen azt tartottam a legcélravezetőbbnek, hogy bizonyos szükséges alapfogalmakat ismertnek tekintsek. Ezek a következők:

- Az adatbázis, adathalmaz, rekord fogalma
- Adatmodellezés, normalizálás, kapcsolatok
- SQL

Mit nyújt ez a rész Önnek?

- **7. fejezet: Adatbázis-kezelés Delphiben**

  Áttekintjük a manapság használatos adatbázis-architektúrákat, majd megismerkedünk a Delphi adatkezelési mechanizmusával.

- **8. fejezet: Adatelérési komponensek**

  Ebben a fejezetben bemutatjuk az adathalmazok kezelésének módszereit, az adathalmazok mezőinek kezelését, a származtatott mezők létrehozását, a fő-segéd űrlapok készítését, és még sok más – Delphi alkalmazásainkhoz szükséges – technikát.

- **9. fejezet: Adatmegjelenítési komponensek**

  Itt foglalkozunk az adatok megjelenítésére specializált komponensekkel.

- **10. fejezet: Könyvnyilvántartó feladat**

  Elkészítünk egy könyvnyilvántartó alkalmazást, melyben begyakorolhatjuk az előző fejezetek anyagát.

- **11. fejezet: SQL utasítások a Delphiben**

  Megismerkedünk az SQL utasítások futtatására kifejlesztett *TQuery* komponens használatával, lehetőségeivel.

- **12. fejezet: Könyvnyilvántartó feladat folytatása**

  Kiegészítjük a 10. fejezetben kitűzött feladatot a szükséges lekérdezésekkel.

- **13. fejezet: Jelentések**

  Bemutatjuk a *QuickReport* komponenscsalád használatát feladatokon keresztül.

- **14. fejezet: Kliens/szerver adatbázis-kezelés**

  Megismerkedünk a kliens-szerver adatbázis-kezelés sajátosságaival egy mintafeladaton keresztül.

7. Adatbázis-kezelés Delphiben

Ebben a fejezetben áttekintjük a manapság használatos adatbázis-architektúrákat, majd megismerkedünk a Delphi adatbázis-kezelési lehetőségeivel, mechanizmusával.
Főbb témáink:

- Fájl-szerver, kliens/szerver és több rétegű adatbázis-kezelési architektúrák
- A Delphi adatbázis-kezelési lehetőségei
- Az adatbázis-kezelési komponensek áttekintése
- Egy egyszerű adatbázisos feladat megoldása

7.1 Az adatbázis-kezelési architektúrák áttekintése

Minden adatbázis-kezelő alkalmazásban három fő funkcionális egységet különböztethetünk meg (7.1. ábra):

- A közvetlen adatkezelés (*Data processing*):

  Az alkalmazásnak ez a része végzi el a tárolt adatok fizikai feldolgozását: állományok nyitása, zárása, indexelések, a lekérdezések optimalizálása és futtatása, új adatok felvitele, meglévők törlése, módosítása, az adatok *cache*-elése, a zárolási konfliktushelyzetek feloldása... Ennek a résznek a megvalósítása függ az adatok tárolási módjától, azaz a használt adatbázis-formátumtól.

- Az alkalmazás-logika[1] (*Business logic*):

  Ez a rész felelős a teljes alkalmazás helyes működéséért: biztosítja az adatok védelmét (felhasználói jogosultságok), elronthatatlanságát (integritását), hatékony és kényelmes kezelését (a műveletek csoportosítása, tranzakció-kezelés[2])... Az alkalmazás-logika feladatkörére és implementálásának módozataira még a 14. fejezetben visszatérünk.

---

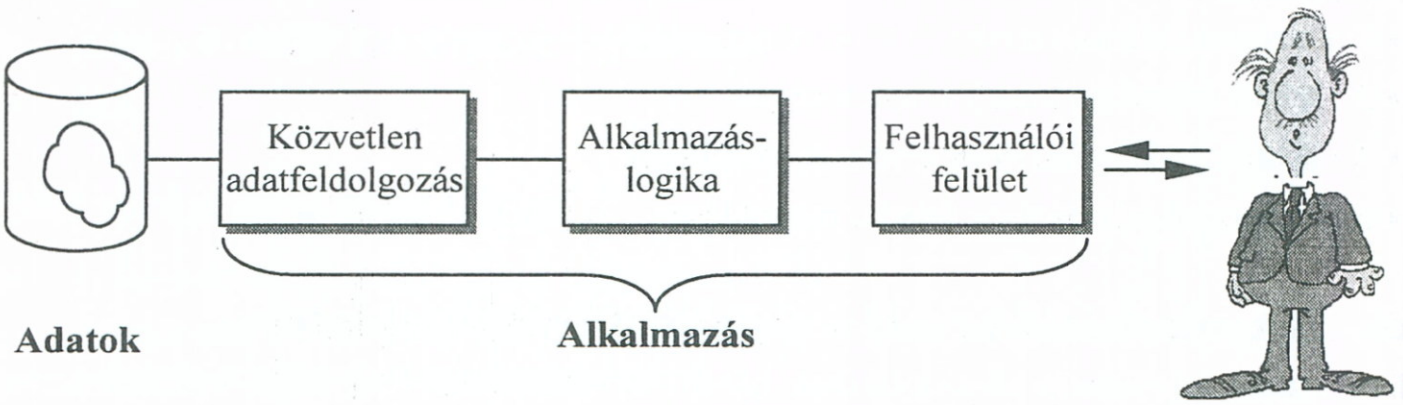
[1] Az angol „*business logic*" kifejezés magyar megfelelőjeként a könyvben az „alkalmazás-logikát" használjuk. A magyar szakirodalomban még az „üzleti logika" és egyéb elnevezésekkel is találkozhatunk, de az összes közül talán ez tükrözi leginkább a feladatát.

[2] Tranzakciónak nevezzük az egyetlen logikai egységet képező adatbázis-kezelési műveleteket. Például egy banki átutalásban a forrás- és célszámla egyenlegeinek módosításai nem választhatók el egymástól (nem lehet „csak félig" átutalni). Vagy a tranzakció összes művelete megtörténik, vagy egyik sem. Egy hiba esetén a teljes tranzakció visszagörgetődik, beáll a tranzakció elindítása előtti konzisztens állapot.

- A felhasználói felület (*User Interface*):

  Ez a rész a felhasználóval való közvetlen kapcsolattartásért felelős. A felületnek minél tetszetősebbnek, barátságosabbnak, és ugyanakkor elronthatatlannak kell lennie. Az elronthatatlanság alatt itt a felhasználói felület elemeinek helyes működésére gondolunk. Az adatok helyességéért nem a felhasználói felület, hanem az alkalmazás-logika felel.

Ehhez a három részhez még hozzátartozik egy utolsó, a tényleges tárolt adat (a fizikai adatbázis). Az alkalmazásnak (programnak, kódnak) azonban a konkrét adatok nem képezik részét.



7.1. ábra. Egy adatbázisos alkalmazás részei

Amikor alkalmazásról beszélünk, nem szükségszerűen egy közvetlenül futtatható EXE állományra kell gondolnunk. Az ábrán látható részek lehetnek DLL állományokban, de lehetnek akár a fizikai adatbázisnak részei is. Nyilvánvaló, van egy EXE, ami az egészet összefogja, futtatja, de ebben nem mindig található meg mind a három egység (csak a legegyszerűbb esetekben).
Az adatok tárolását különböző formátumú állományokban valósítjuk meg. Vannak olyan típusú állományok, melyek ténylegesen csak az adatokat tárolják, és vannak olyanok is, melyekbe már bizonyos alkalmazás-logikához tartozó szabályokat, megszorításokat is be lehet építeni.

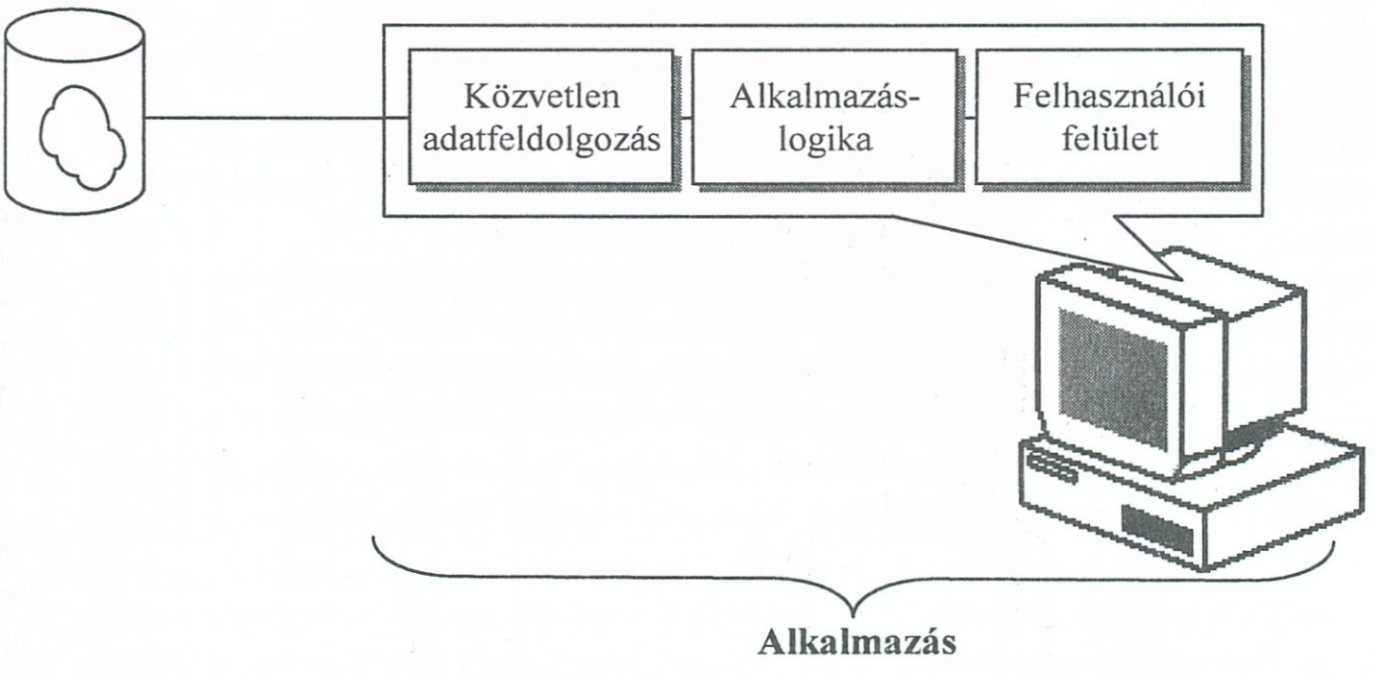
A legtöbb adatbázisos alkalmazásra az jellemző, hogy megosztott adatokon dolgozik, vagyis a fizikai adatbázist nem egy felhasználó használja egyedül, kizárólagosan, hanem egyszerre több felhasználó, több számítógépről párhuzamosan kezeli ugyanazokat az adatokat. Ezért az adatok mindig egy kiemelt szerepű gépen találhatók. De mi van az alkalmazással? Minden gépen teljes egészében jelen van, vagy csak egy része van a célgépen, másik része pedig az adatok mellett? Az alkalmazás három része fizikailag is külön gépeken helyezkedhet el. Attól függően, hogy az alkalmazás részei hány gépre, és hogyan lettek elosztva, három fő adatbázis-kezelési architektúráról beszélhetünk:

- Fájl-szerver (*File server*) architektúra
- Kliens/szerver (*Client/Server*) architektúra
- Több rétegű (*Multi-tier*) architektúra

Tulajdonképpen mindhárom fogalom túlmutat az adatbázis-kezelés keretein, ezek fellelhetők más alkalmazási területeken is, mint a különböző levelezőrendszerek, Web böngészés, hálózat-adminisztráció, fájl-kezelés (helyi hálózat, ftp) stb. A következőkben az adatbázis-kezelés szemszögéből mutatjuk be ezen architektúrákat.

7.1.1 Fájl-szerver (*File server*) architektúra

A fájl-szerver architektúrában az alkalmazás mindhárom része egyetlen gépen helyezkedik el (7.2. ábra).



7.2. ábra. A fájl-szerver architektúra

Ameddig az adatok és az alkalmazás ugyanazon a számítógépen találhatók, addig ez a megoldás nem rendelkezik különösebb hátránnyal: adatkezeléskor a szükséges adatállomány a merevlemezről betöltődik a memóriába, ott feldolgozásra kerül, majd az esetleges módosulások visszaíródnak a lemezre.

A valós igények megkövetelik a közös, „központi" gépen elhelyezkedő adatokon történő többfelhasználós munkát. A feldolgozandó adatok a hálózaton keresztül mindig átkerülnek a célgépekre, ahol a felhasználók saját gépükön használják ezeket. Ebből fakad e technika nagy hátránya: a teljes adathalmaz tekintélyes méreteket ölthet, ezek átvitele akár többször is „leterhelheti" a hálózatot, vagyis egy közös erőforrást. Így az egyes felhasználók munkája gyakran a többiek munkáját gátolja. Ez azért van, mert legtöbbször az adatátviteli láncolatok leglassúbb eleme – „szűk keresztmetszete" – nem más, mint maga a hálózat. Itt

fontos szerepet játszanak a közös erőforrások (központi gép lemez→memória, memória→hálókártya), valamint az egyéni erőforrások (felhasználók gépeinek hálókártya→memória) átviteli sebességei. Szoftverileg, az adatgépen egy fájl-szerverre van szükség, mely elérhetővé teszi az adatállományokat a hálózati felhasználók számára (például Windows for Workgroups vagy későbbi, Novell Server). Vagyis hálózati vonatkozásban nincs semmi különbség az efféle adatkezelés és egy egyszerű osztott hálózati fájlelérés között. Ezt a technikát alkalmazza a dBase, FoxPro és Clipper (állományaiknak kiterjesztése DBF), a Paradox (DB), a Visual Basic és Access[1] (MS Jet adatgép, MDB), vagy egy egyszerű tetszőleges nyelven megírt adatkezelő-rendszer. Ez azt jelenti, hogy amikor ilyen típusú adatokhoz szeretnénk hozzáférni Delphiből vagy más adatkezelőből (akár ODBC[2]-n keresztül is), minket is érinteni fognak ezek a hátrányok.
Az említett hátrányok miatt szükség volt más architektúrák kidolgozására. Az új technikák jellemzőiből kiindulva, utólag a fájl-szerver technikát **egy rétegűnek (*single-tier*)** nevezték el (mivel minden feladatot egy gép végez el).

7.1.2 Kliens/szerver (*Client/Server*) architektúra

Ebben a technikában az alkalmazás két részre bomlik: az adatok közvetlen kezelése az adatok tárolásáért felelős központi gépen történik egy **adatbázis-szervernek** nevezett szoftver által. Célszerű ugyanakkor itt elhelyezni az alkalmazás-logikának nagyobbik részét is, ez úgymond az adatbázis részévé válik. Ez azért lehetséges, mert az itt használatos adatbázis-formátumok támogatják bizonyos programrészek tárolását (triggerek, tárolt eljárások formájában – bővebben lásd a 14. fejezetben). Ezek implementálják az alkalmazás-logika szerver oldali részét. A kliens gépeken már csak az adatbázisba be nem épített alkalmazás-logika és a felhasználói felület kerül (7.3. ábra).

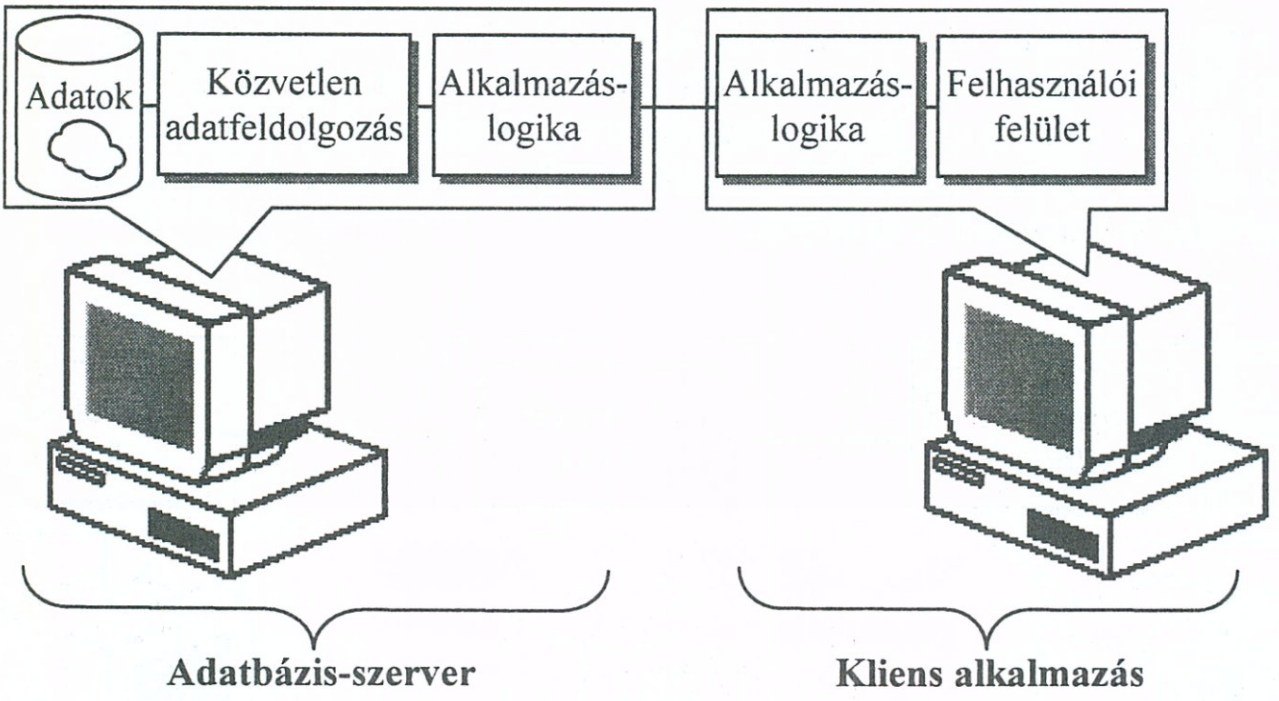
Ezt a modellt még **ügyfél-kiszolgálónak** is nevezik: az ügyfél gépén futó kliens alkalmazás lekérdez bizonyos jellemzőjű adatokat, a kérés „elutazik" a szerverhez, aki ezt a leválogatást elvégzi (nála van az adatfeldolgozó egység), és az eredményt (figyelem, csak az eredményt!) visszaküldi az ügyfélhez. Tehát a hálózaton nem a feldolgozandó adatok közlekednek (akár oda-vissza), hanem előbb elmegy a szerverhez a megfelelő adatfeldolgozó parancs, és csak a parancs eredménye (ha van) utazik vissza a hálózaton a kliens géphez. Emiatt a hálózati átvitel teljesítménye sokat javul. Ráadásul az adattároló gép (most már „szerver") nagyteljesítményű hardverét (erre az előző esetben is szükség volt a közös használat miatt) most egy nagyteljesítményű szoftverrel támogatjuk, ami kifejezetten az adatkezelésre lett optimalizálva. Ilyenek a nagyobb adatforgalommal rendelkező vállalatoknál megtalálható adatszerverek: Informix, Sybase, Oracle, MS SQL Server és nem utolsó sorban a Borland-féle InterBase, amire könyvünkben többször hivatkozunk

---

[1] Az Access adatkezelési technikája fejlettebb a többi felsorolt fájl-alapú rendszerekénél (dBase...), de távol áll az igazi kliens/szerver adatbázis-kezeléstől.

[2] ODBC = *Open Database Connectivity*. Ez egy Microsoft által kifejlesztett rutincsomag, mely a különböző formátumú adatbázisok egységes kezelését teszi lehetővé (bővebben lásd kicsit később).

(lásd 14. fejezet). Ez utóbbinak egy ún. „lokális" változatát – vagyis hálózatot nem támogató, így kifejezetten az egyetlen gépen történő alkalmazásfejlesztés célját szolgáló változatot – a Delphivel együtt forgalmazzák.



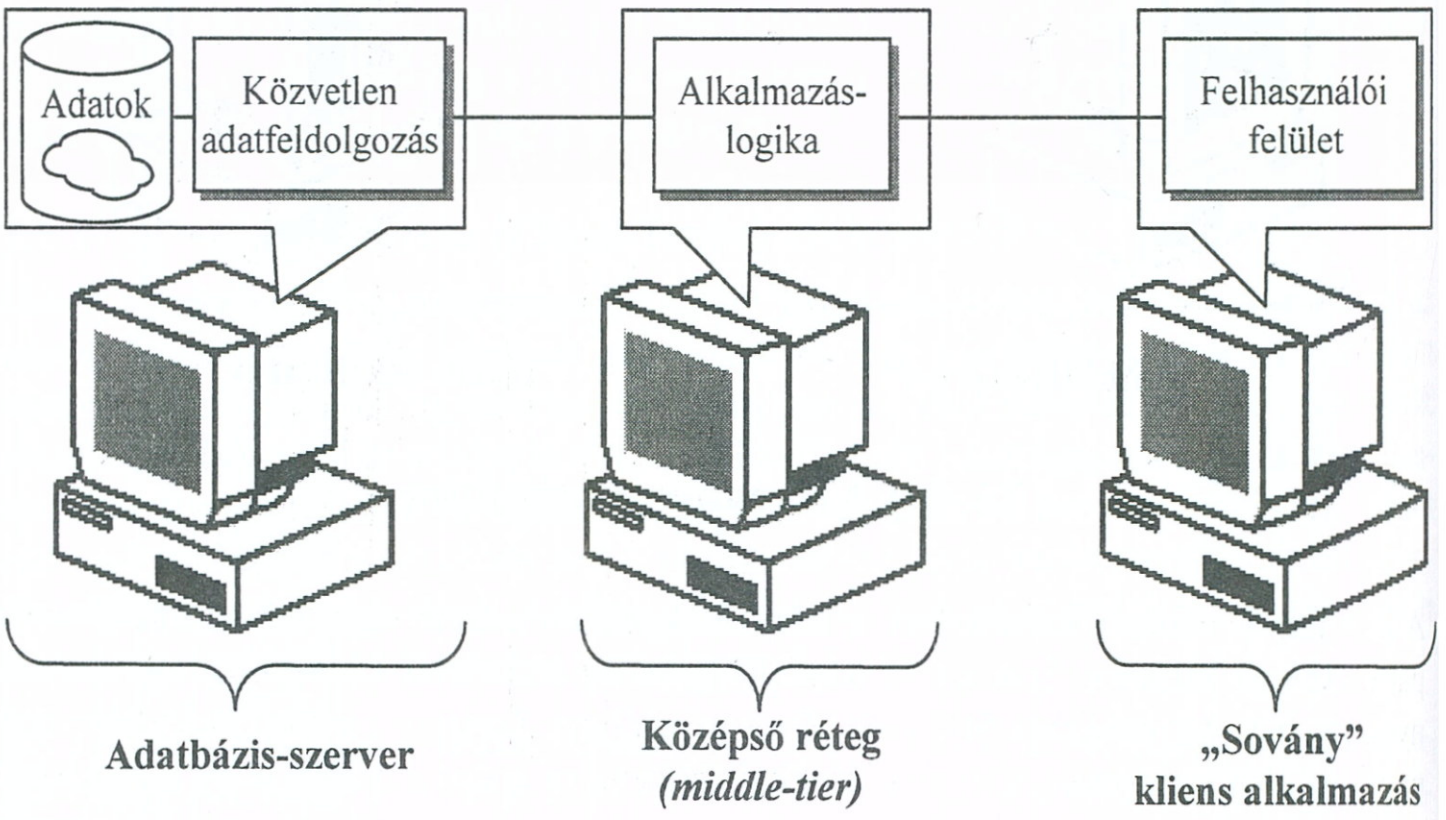
7.3. ábra. A kliens/szerver architektúra

Amint látjuk, ebben a technikában az adatfeldolgozást a szerver végzi a kliens alkalmazás parancsainak hatására. Ezeknek az adatfeldolgozó parancsoknak kidolgoztak egy szabványos nyelvet, ezt nevezzük **SQL**-nek (***S**tructured **Q**uery **L**anguage* = strukturált lekérdező nyelv). Tehát a kliensek SQL utasításokkal késztetik rá a szervert arra, hogy a megfelelő műveletet (-eket) ott helyben elvégezze, és az eredményt visszaküldje. Az adatfeldolgozás központosításának rengeteg előnye van: központilag, egységesen ellenőrizhetjük le az adatok helyességét és a felhasználók jogosultságait; zárolások által könnyen és hatékonyan vezérelhetjük a párhuzamos adatelérést stb. Sajnos a valóságban nem ennyire „rózsaszínű" a helyzet. Szép gondolat az adatellenőrzés és feldolgozás központosítása, de a bonyolultabb esetekben ez nem valósítható meg teljes egészében a szerveren. Ha valamit nem lehet, túl bonyolult, vagy nem célszerű a szerveren létrehozni, az a kliensre marad. Vagyis a felhasználói felület bizonyos logikával is fog rendelkezni, más szóval előfordulhat, hogy az alkalmazás-logika egy része a kliensen marad. Ez több okból is hátrányos, mivel így a szerver-oldali adatfeldolgozás jó néhány előnye nem garantált. Például nem tartható egy kézben a biztonsági rendszer és az adatellenőrzés. Mivel a szerver minden olyan klienst „szívesen fogad", aki ért a nyelvén (tud „SQL-ül"), elképzelhető, hogy például ODBC segítségével egy felhasználó közvetlenül a szervert kezeli. Ezzel megkerüli a kliens oldali logikát, vagyis kiiktatja a teljes alkalmazás egy részét, valószínűleg nem jó szándékkal.

Mivel ebben a technikában az alkalmazás két részre bomlik, ezt az architektúrát később **két rétegűnek (*two-tier*)** nevezték el.

7.1.3 A több rétegű (*multi-tier*) adatkezelés

Ebben a technikában az alkalmazás részei kettőnél is több gépen helyezkednek el. A 7.4. ábra egy tipikus három rétegű alkalmazás szerkezetét mutatja be. A több rétegre való eloszlás további előnyökkel jár (munkamegosztás, adminisztráció könnyítése...). Kliens oldalon már ténylegesen csak a felhasználói interfész található, ezért ezt az alkalmazást **„sovány" (*thin*) kliensnek** is nevezzük.



7.4. ábra. Az adatbázis-kezelésben a leggyakrabban használt három rétegű alkalmazás szerkezete

Megjegyzés:
A következő 6 fejezeten keresztül általános, mindhárom architektúrában érvényes Delphi adatbázis-kezelési technikákat ismerhetünk meg. A kliens/szerver sajátosságokat a 14. fejezetben, a három rétegű fejlesztés lehetőségeit pedig a 19. fejezetben mutatjuk majd be.

7.2 A Delphi adatbázis-kezelési lehetőségei

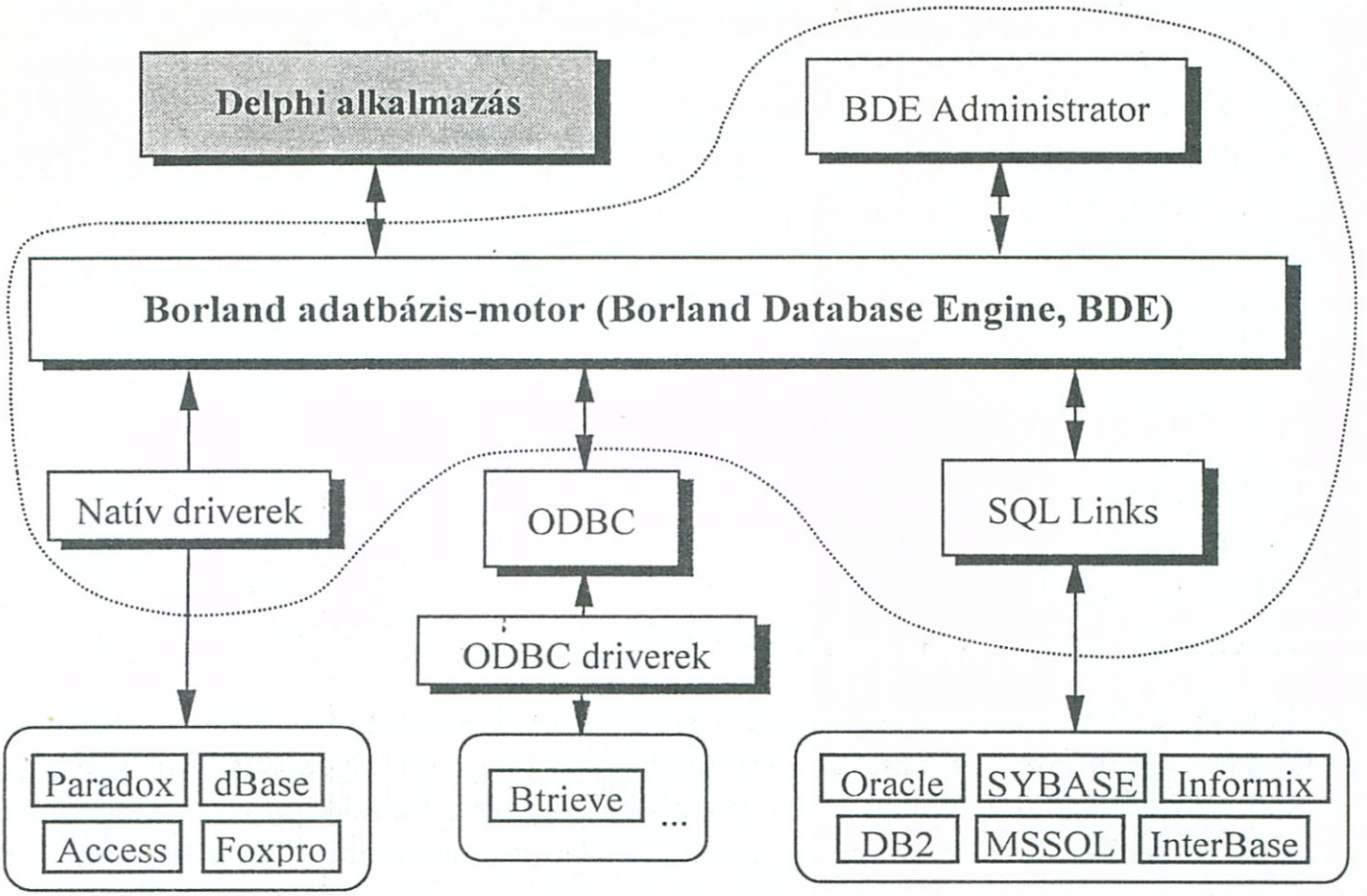
A Delphiből a következő formátumú adatbázisokat érhetjük el:

- Natív driverekkel:
  - dBase, Foxpro (*.DBF)
  - Paradox (*.DB)
  - Access (*.MDB)
  - DB2
  - InterBase
  - Microsoft SQL
  - Oracle
  - SYBASE
  - Informix
- ODBC-n keresztül:
  - Minden, amihez telepítünk ODBC drivert (Btrieve, Excel...)

Delphiben az adatbázisok kezelése speciális komponensek segítségével történik. Alkalmazásainkban a különböző formátumú adatbázisokat egységesen, ugyanazokkal a komponensekkel érjük el. Ezen a szinten nincs különbség például a Paradoxban és az Oracleben tárolt adatok kezelése között (csupán a komponensek egy jellemzője utal a konkrét adatbázis formátumára és elhelyezkedésére). A beépített osztálygyűjteményt **IDAPI**-nak (**I**ntegrated **D**atabase **A**pplication **P**rogramming **I**nterface) is szokták nevezni.
A komponensek metódusai a beépített adatbázis-motor *(**B**orland **D**atabase **E**ngine,* a továbbiakban BDE*)* rutinjait használják (lásd 7.5. ábra). Tehát **a BDE egy egységes felületet (rutincsomagot) biztosít a különböző formátumú adatbázisok Delphiből történő kezelésére.**

Elemezzük most a 7.5. ábra alsó felét! Hogyan éri el a BDE a különböző formátumú adatbázisokat?

A fájl-szerver architektúrában, ahol a közvetlen adatkezelés a Delphi alkalmazás része, ezt a feladatot a BDE látja el bizonyos adatbázis-formátum specifikus driverek segítségével. Ahhoz, hogy az adatbázis-motor egy adott formátumban (például Paradoxban) tárolt adathalmazt kezelni tudjon, szükséges a megfelelő meghajtó (például a Paradox esetén: IDPX32.DLL); ez tartalmazza azokat a rutinokat, amelyek kifejezetten a Paradox táblák kezelését valósítják meg. Egyes meghajtók már eleve a Delphivel érkeznek (mint a Paradox, dBase, Access...driverei), ezek az ún. **natív meghajtók**. Ha egy olyan adatbázist szeretnénk kezelni Delphiből, amihez van natív driver, akkor a Delphi adatkezelési komponenseink meghívják a BDE rutinokat, ezek pedig közvetlenül hívják a megfelelő driver rutinjait (7.5. ábra bal alsó része). Ha viszont egy olyan formátumú adatbázist akarunk Delphiből elérni, amihez nincs beépített meghajtó (például Btrieve), akkor az ODBC-t kell segítségül hívnunk, és ezen keresztül az ún. **ODBC drivert** fogjuk az adatok elérésére használni.



7.5. ábra. Adatbázisok kezelése Delphi alkalmazásból[1]
(A „babapiskóta" a Delphi határát jelzi)

Az ODBC egy Microsoft által bevezetett rutincsomag, mely – akárcsak a BDE – egy egységes felületet biztosít a különböző formátumú adatbázisok kezelésére. (Az ODBC felület rutinjai általánosan használhatók, például Wordből, Excelből, Accessből..., ugyanakkor a BDE egységes felületét csak a Delphi alkalmazásainkból használhatjuk.[2])
Egy adott formátumú adatbázis eléréséhez az ODBC-nek is szüksége van – akárcsak a BDE-nek – az adott formátumnak megfelelő meghajtóra, az ún. ODBC driverre. Ezek a meghajtók származhatnak az adatbázis-gyártójától, egy *third-party* meghajtó-forgalmazótól (például www.intersolv.com), vagy akár az MS Officeból. Ha tehát Delphiből egy olyan adatbázist szeretnénk elérni, amihez nem tartozik natív driver, akkor az ODBC-t hívjuk segítségül: ekkor a Delphi komponenseink hívják a BDE rutinokat, ezek meghívják az ODBC felület rutinjait, és majd csak ezek hívják meg a megfelelő ODBC driver eljárá-

[1] Az ábra a Borland Delphi for Windows 95 & Windows NT: Database Application Developer's Guide című könyvben található ábra alapján készült.

[2] A BDE abban különbözik az ODBC-től, hogy a BDE adatbázisgépet is tartalmaz, míg az ODBC csak az általános célú felületet tartalmazza.

sait (7.5. ábra középső rész). Ebben a megoldásban tehát az adatok eléréséhez még egy szint szükségeltetik. Ezért van az, hogy **a natív driverek használata gyorsabb adatelérést biztosít, mint az ODBC drivereké**.

A kliens/szerver architektúrában, mivel itt a közvetlen adatkezelés a szerveren folyik, a Delphi komponensek metódusait a BDE-nek az adatbázis-szervernek megfelelő SQL utasításokká kell alakítania, majd az utasításokat továbbítania kell a szerver felé. Az SQL utasítások előállítását a BDE az ún. **SQL Links drivercsomag** segítségével végzi el (7.5. ábra jobboldali része). A kliens-szerver architektúránál is megoldást biztosít az ODBC, természetesen csak ha rendelkezésére áll a megfelelő SQL Links meghajtó.

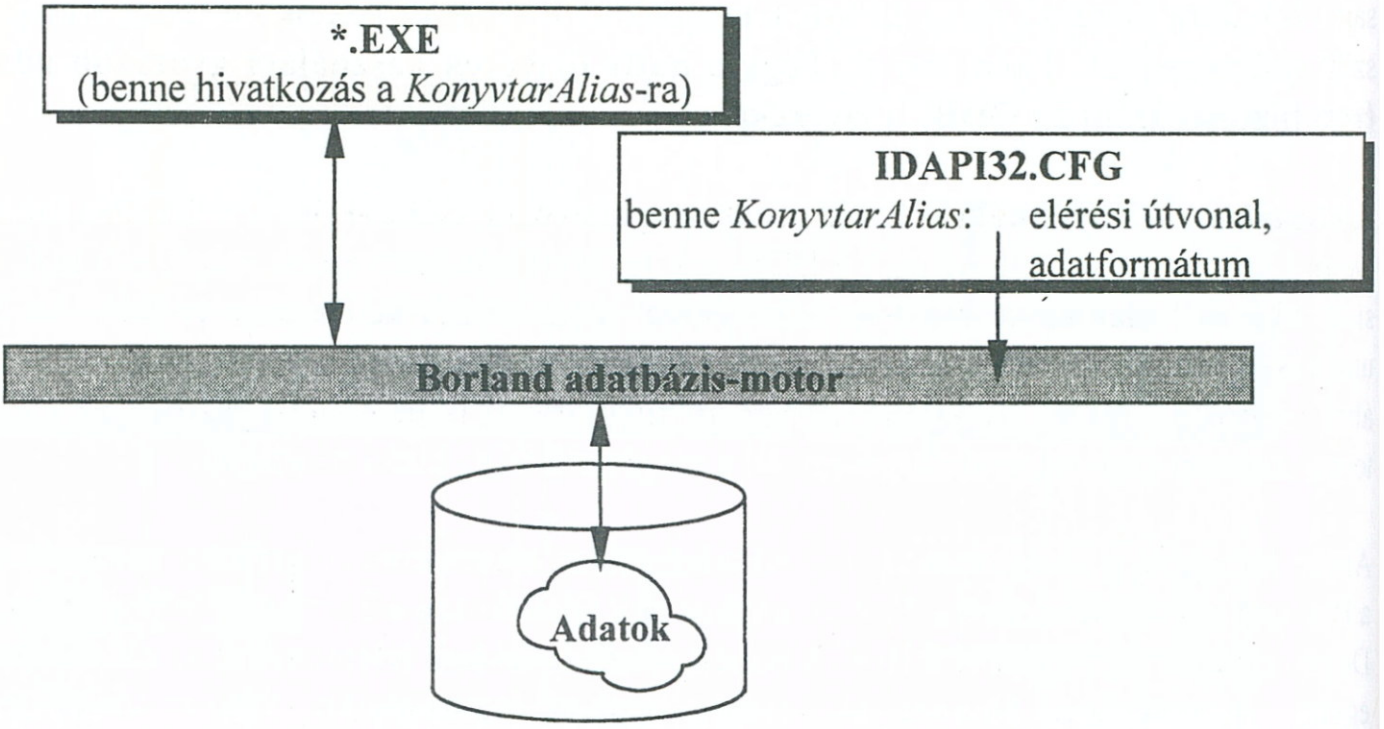
Amint láthattuk, a Delphiben fejlesztett adatbázisos alkalmazásaink erősen támaszkodnak a Borland adatbázis-motorra. Emiatt, amikor egy adatbázisos alkalmazást feltelepítünk egy Delphi nélküli gépre, akkor a BDE használt részét is fel kell telepítenünk az alkalmazással együtt. A szükséges állományok kiválogatásában segítségünkre van az *InstallShield Express* telepítőprogram-készítő. (Bővebben lásd a 17. fejezetben.)

7.3 Az álnév (Alias)

Egy adatbázisos alkalmazásban az adatok helyét és formátumát valamilyen módon specifikálnunk kell. Ne felejtsük el, hogy fejlesztés közben az adatok egy konkrét elérési útvonalon helyezkednek el, de átadás után ez a hely megváltozhat.

Tételezzük fel, hogy beégetjük programunkba a konkrét elérési útvonalat (az EXE állományban erre hivatkozunk). Ekkor az alkalmazás – egy másik gépen történő telepítése után – még mindig a régi helyen keresné az adatokat. Erre a problémára csak egy megoldás létezik: fordítsuk újra az alkalmazást a célgépen, immár a jó elérési útvonallal. Ehhez viszont a célgépen szükség van a Delphi keretrendszerre és természetesen az alkalmazás forrásállományaira is. Ez nem a járható út!

A megoldás az álnevek használatában rejlik. Hozzunk létre fejlesztés közben egy álnevet: ezt az adatbázis-motor konfiguráló programjával, a *BDE Administrator*ral, vagy a *Database Explorer* program segítségével tehetjük meg (Az IDAPI32.CFG konfigurációs állomány tartalmazza a létező álneveket, innen olvassa ki a BDE a beállításokat). Az álnév tartalmazza az adatok elhelyezésére és formátumára vonatkozó információkat. Alkalmazásunk lefordítása után, az EXE állományban csak az álnévre találunk hivatkozásokat. A célgépen való telepítés végén elég az álnevet átállítani az új elérési útvonalra (ezt a feladatot is a telepítőprogram látja el, lásd 17. fejezet), és szoftverünk máris futtatható állapotba kerül. Ezt szemlélteti a 7.6. ábra is.



7.6. ábra. Az álnév használata a Delphi alkalmazásainkban

Foglaljuk össze az álnevek előnyeit:

- Nem kell minden adatáthelyezés után újrafordítanunk az alkalmazást.
- Tervezéskor elég egy rövid álnevet begépelnünk a megfelelő helyekre, nem kell a hosszú elérési útvonallal bajlódnunk.
- SQL Serverek esetén az álnév egyéb, sajátos információkat is tartalmazhat (például a felhasználó nevét (*User Name*), szerver neve (*Server Name*) stb.).
- Az álnév használata alkalmazásunkat nemcsak az adatbázis helyétől függetleníti, hanem a konkrét formátumától is. Egy álnevekre hivatkozó alkalmazás elvileg működőképes marad az adatbázis formátumának megváltoztatása után is.

Tehát az álnév egy munkánkat megkönnyítő hivatkozási rendszer. Alapja az, hogy az álnévhez tartozó információk nem lesznek beégetve az alkalmazásba. Ehelyett ezek az információk egy „járulékos" állományba kerülnek, mely könnyen szerkeszthető akár egyszerű eszközökkel is.

Hasonló elven működik az ODBC is. Az alkalmazás ott is csak álneveket (DSN = **D**atasource **N**ame) tartalmaz. A különbség csak annyi, hogy az ODBC esetén az álnevek leírásai a regisztrációs adatbázisban (*registry*) találhatók.

Hozzunk létre egy új álnevet, mely a Delphi mintaadatbázisára mutat!
Ennek érdekében indítsuk el a BDE Administrator (régebbi verziókban Database Engine Configuration) segédprogramot. A megjelenő ablak *Configuration* (régebbi verziókban *Drivers*) oldalán megtekinthetjük a feltelepített natív és ODBC meghajtókat. A *Databases* (*Aliases*) oldalon a létező álnevek listáját láthatjuk.
Hívjuk meg az *Object/New* menüpontot (*New Alias* gombot). Az előbukkanó párbeszédablakban ki kell választanunk az új álnév adatainak formátumát (régebbi verziókban itt kell megadnunk az álnév nevét is). A mintaadatbázis Paradox típusú, így most a *Standard* formátumot válasszuk (Figyelje meg, hogy a formátumok listájában az előbbiekben megtekintett meghajtók szerepelnek!). A párbeszédablak bezárása után már csak az álnév adatainak elérési útvonalát (*Path = ...DELPHI\DEMOS\DATA*), valamint az álnév nevét kell megadnunk.
Figyelje meg, milyen egyéb információkat tartalmaz például az IBLOCAL álnév. Ez a Delphi Interbase mintaadatbázisának álneve. Az Interbase adatbázisok kezelésével bővebben a 14. fejezetben foglalkozunk.

7.4 A Delphi adatbázis-kezelést elősegítő segédprogramjai

A Delphihez számos segédprogram tartozik. Nézzük most a legfontosabbakat az adatbázis-kezeléssel kapcsolatosak közül:

- **BDE Administrator** (régebbi változatokban: *BDE Configuration*): ez az adatbázismotor konfigurációs programja. Fontosabb funkciói:
  - Álnevek létrehozása, módosítása, törlése
  - Új ODBC adatforrások hozzáadása

  A *BDE Administrator* segédprogram az IDAPI32.CFG (Delphi 1-ben: IDAPI.CFG) állományba ír, ezen keresztül kommunikál a BDE-vel.
- **Database Explorer** (Delphi 1-ben nincs megfelelője):

  Ez egy nagyon hasznos segédprogram, sajnos csak a 32 bites változatok Client/Server csomagjában található meg. A következőkre van benne lehetőség:
  - Álnevek kezelése
  - Adatbázisok szerkezetének (táblák – azon belül mezők, indexek, hivatkozási integritás[1] stb. –, nézetek, tárolt eljárások...) és tartalmának megtekintése, módosítása

[1] Hivatkozási integritásnak (*Referential Integrity*) nevezzük azt a szabályt, mely szerint egy idegen kulcsban nem létezhet olyan érték, amelyhez nem párosul megfelelő elsődleges kulcsérték a vele kapcsolatban álló egyoldali táblában. Például van egy Alkalmazottak és egy Városok táblánk. Az alkalmazottakról tároljuk a szülővárost is, az azonosító formájában. Az alkalmazottaknál csak olyan városazonosítót adhatunk meg, mely a Városok táblába előzőleg felvitt városra hivatkozik.

- SQL lekérdezések begépelése és lefuttatása
- Adatszótárak kezelése (lásd 14. fejezet)

- **Data Migration Wizard** (más változatokban: Data Pump Expert, Delphi 1-ben nincs megfelelője): különböző adatbázisok közötti metaadat (az adatbázis szerkezeti információja) és adat áthelyezésére használható.
- **DataBase Desktop**: helyi adatállományok kezelésére fejlesztették ki. (Elvileg, bizonyos megkötésekkel, adatbázis-szerverek tábláinak kezelésére is alkalmas, de a korlátozások miatt a gyakorlatban erre nemigen használják.)
- **SQL Monitor** (Delphi 1-ben nincs megfelelője): az SQL lekérdezések nyomkövetési segédprogramja. A BDE által az SQL szerver felé küldött SQL utasítások megfigyelésére használhatjuk.
- **Server Manager**: az Interbase adatbázis-szerver karbantartó és felügyeleti programja. Alkalmas felhasználók (*Users*) kezelésére, adatbázisok mentésére...
- **Windows ISQL**: az Interbase adatbázisok SQL parancsok általi közvetlen kezelésére alkalmas (ablak, amiben SQL parancsokat lehet osztogatni). A 14. fejezetben mi is használni fogjuk.

7.5 Adatbázis-kezelési komponensek

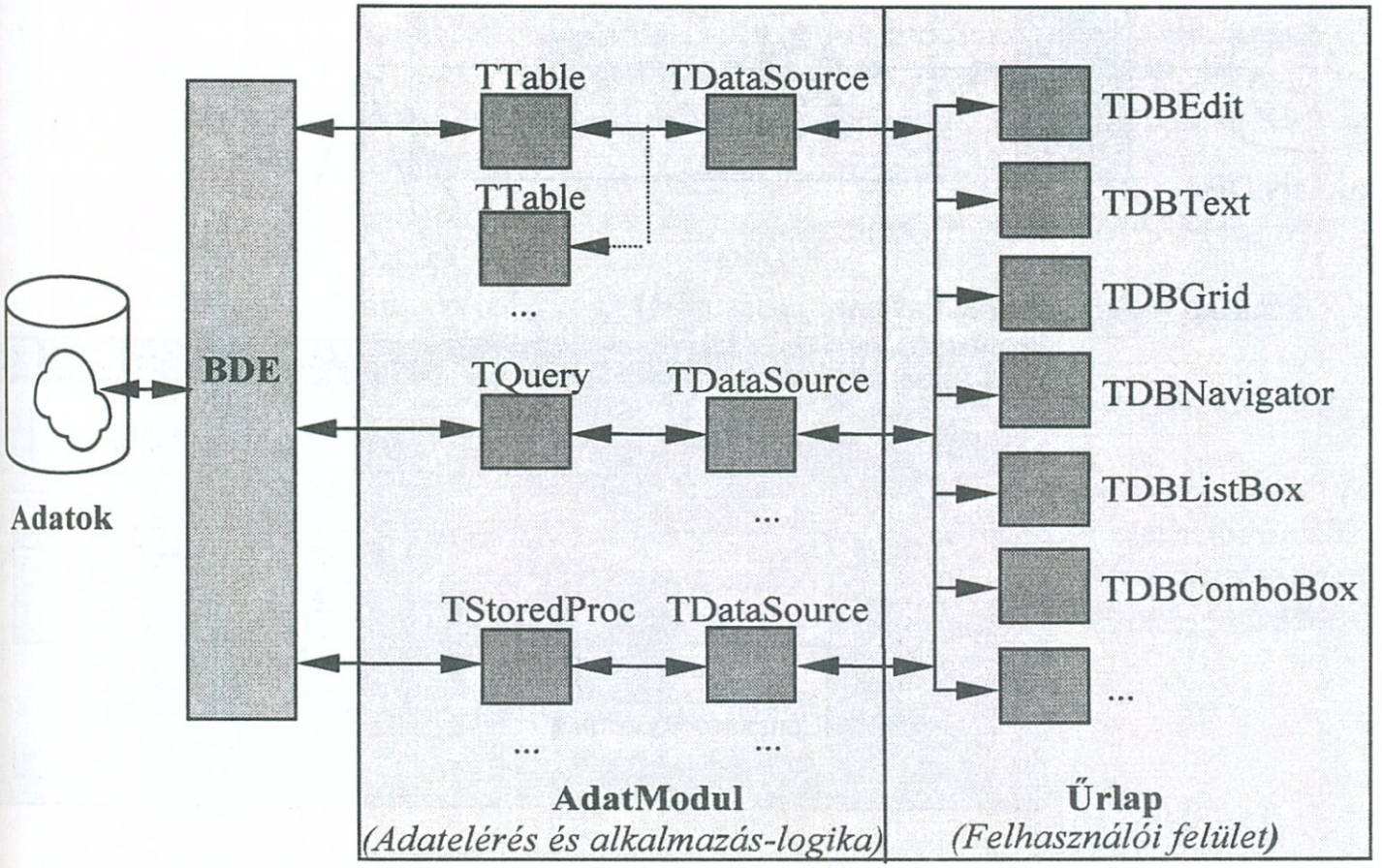
Delphiben az adatbázis-kezelést elősegítő komponenseket két fő kategóriába sorolhatjuk:

- **Adatelérési (*Data Access*) komponensek**: adatbázisok (*TDatabase*), táblák (*TTable*), lekérdezések (*TQuery*), tárolt eljárások (*TStoredProc*)... kezelését teszik lehetővé. Az adatelérési komponensek adatainak megjelenítésére speciális adatmegjelenítési komponenseket kell használnunk.
- **Adatmegjelenítési (*Data Controls*) komponensek**: csupán megjelenítési célokra kifejlesztett komponensek, melyek az adathozzáférést biztosító komponensekhez kapcsolódnak (az adatforrásukon keresztül), így ezek pillanatnyi állapotát tükrözik. Például egy tábla (*TTable*) egy adott mezőjére irányított *TDBEdit* komponens a tábla aktuális rekordjából az adott mező értékét jeleníti meg. Minden adatmegjelenítési komponens típusazonosítójában a *DB* előtag azt mutatja, hogy egy adatforrásból „táplálkozik". (*MS Access*ben ezeket „kötött vezérlőelemeknek" nevezzük.)

  Adatmegjelenítési komponensek: szerkesztődoboz (*TDBEdit*), címke (*TDBText*), rács (*TDBGrid*, egy egész adathalmaz megjelenítésére alkalmas), navigátorsor (*TDBNavigator*, egy adathalmaz rekordjai közötti lépegetésre), kombinált lista (*TDBComboBox*), kép (*TDBImage*)...

Az adathozzáférési komponensek egymáshoz is kapcsolódnak (kapcsolódhatnak), így együttesen alkotják az alkalmazás adatelérési- és alkalmazás-logikáját (*business logic*): ide építhetünk be számított mezőket, valamint további adatellenőrzéseket, adatszűréseket... (Például egy könyvtárból maximum 5 könyvet lehessen csak egyszerre kikölcsönözni.) A Delphi 32 bites verzióiban ezeket az elemeket egy ún. **adatmodulra (*DataModule*)** he-

lyezzük el. Ez tervezési időben egy külön ablakként jelenik meg, futás közben viszont láthatatlanná válik. Szerepét – az adatokhoz való hozzáférés biztosítását – a háttérből látja el.

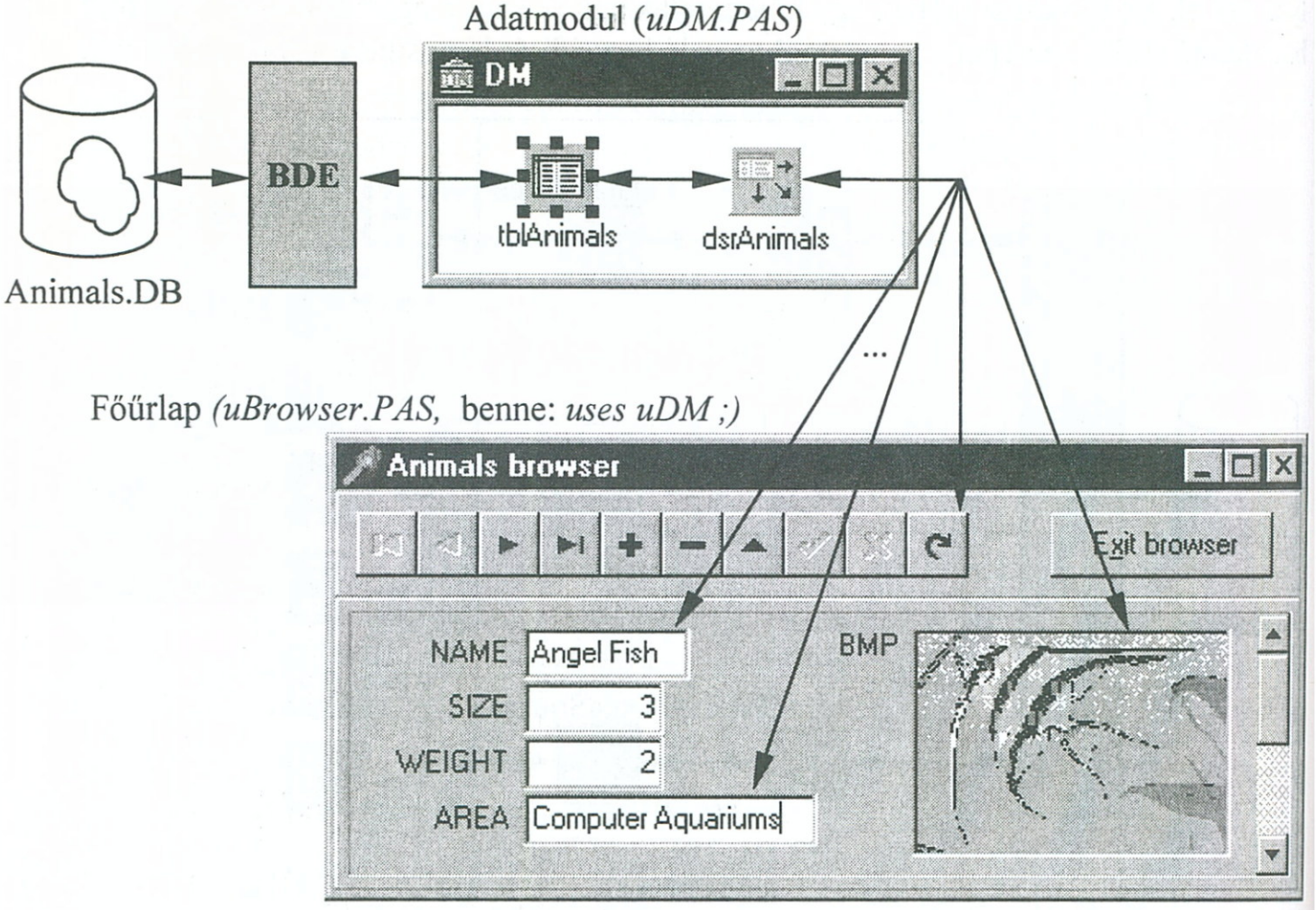


7.7. ábra. Az adatelérési és megjelenítési komponensek kapcsolata[1]

Az adatmegjelenítési komponensek a felhasználói felület (*user interface*) elemei; azt mutatják, amit a kapcsolt adathozzáférési komponenstől kapnak. Helyük azon az űrlapon van, amelyiken az adatokat meg szeretnénk jeleníteni. Mindnyájan rendelkeznek egy ún. *DataSource* jellemzővel, melyen keresztül kapcsolódnak az adatmodulon valamely adatforrásához. Ennek az adatforrásnak a tartalmát jelenítik tehát meg, íráskor pedig ezt állítják át.

Leendő Delphi alkalmazásainkban (kódból) egy tárolt adatot **sohasem** az adatmegjelenítés felől fogunk megközelíteni (például *DBEdit1.Text*), hanem értékét a megfelelő adatforrásból fogjuk kiolvasni, szükség esetén itt fogjuk átírni. Az adatmegjelenítési komponens mindig frissül, ha az adatforrásban átírunk „alatta" egy adatot.

[1] Az ábra a Borland Delphi for Windows 95 & Windows NT: Database Application Developer's Guide című könyvben található ábra alapján készült.



7.8. ábra. Egyszerű adatbázisos alkalmazás

A 16 bites Delphiben az adatelérési logika és felhasználói felület elkülönítése még nem volt ennyire egyértelmű. Ott még nem létezett adatmodul, így az adathozzáférési komponenseket az űrlapokra kellett elhelyeznünk. Ez egyaránt megnehezítette az alkalmazás megtervezését és implementációját is. Ugyanazt a logikai funkciót ellátó adatforrást minden űrlapon el kellett helyeznünk, ahol a tartalmára szükségünk volt. És ha netalán mindkét ablakban meg szerettük volna jeleníteni ugyanazt a számított mezőt, akkor dolgozhattunk volna érte kétszer. Szerencsére a 32 bites Delphi verziókban ez már nem így van (lásd 7.8. ábra).

Delphi 1-ben az adatmodul egy közönséges űrlappal helyettesíthető. Az adatelérési komponenseket egy űrlapra helyezzük el, az űrlap egységét pedig minden más, adatot használó űrlap egységébe építsük be. Természetesen az adatmodul szerepét játszó űrlapot futáskor ne jelenítsük meg, de ugyanakkor vigyázzunk arra, hogy létrejöjjön (*auto-create* legyen). Ez a megoldás annyiban rosszabb, mint az igazi adatmodulos, hogy itt tervezési időben még nem hivatkozhatunk az adatmodul szerepű űrlap komponenseire. Az adatmegjelenítési komponensek *DataSource* jellemzőit így futási időben, kódból kell beállítanunk.

7.6 A TDataModule osztály

- Helye az osztályhierarchiában: *TObject/TComponent/TDataModule*
- Szerepe: alkalmazásunk futási időben láthatatlan komponenseinek gyűjteményét tartalmazza (például *TTimer, TOpenDialog... TTable, TDataSource...*). Tervezéskor egy vizuális konténerként viselkedik (akárcsak egy űrlap), futási időben viszont az adatmodul láthatatlan. Előnyei:
  - A ráhelyezett komponenseket az alkalmazás űrlapjai megosztva használhatják nemcsak futási, hanem már tervezési időben is.
  - Adatbázisos alkalmazások esetén megvalósítja az adatelérési logika és a felhasználói felület elkülönítését.
  - Különböző rutinokat (metódusokat) is implementálhatunk benne, ezek is hívhatók lesznek az egész alkalmazásban.
  - Az elkészített adatmodult mintaként kimenthetjük (a *gyorsmenü Add To Repository* parancsával), így ez más alkalmazások számára is elérhetővé válik. Ha egy későbbi programban szükség lenne egy ilyen szerkezetű adatmodulra, akkor a *File/New...* menüpont segítségével a *DataModules* oldalról ezt kiválaszthatjuk.
- Két gyakrabban használt eseményjellemzője van:
  - **OnCreate**
  - **OnDestroy**

  Az *OnCreate*-ben szoktuk szükség esetén a rajta található komponenseket inicializálni (a táblakomponenseket itt nyitjuk meg). Az *OnDestroy* eseménybe végső tevékenységeket építhetünk be (itt zárjuk be a táblákat).

Figyelem! Ajánlatos, hogy az adatmodul *auto-create* legyen, és elsőnek jöjjön létre, még az űrlapok előtt[1]. Ez akkor nagyon fontos, amikor egy űrlap *OnCreate* eseményében már hivatkozunk az adatmodul részeire (például kiolvassuk egy tábla mezőit). Az űrlap létrehozásakor tehát, az adatmodulnak már léteznie kell, egyébként a már jól ismert „Access Violation at Address..." hiba következik be (mivel a még nem inicializált adatmodul-objektum tábla-mezőjére hivatkozunk).

Az adatmodul használata

Alkalmazásunkban új adatmodult a *File/New Data Module* menüponttal hozhatunk létre. Létrehozása után rögtön nevezzük el (például *Name= DM*), majd mentsük is le állományba (például *uDM.PAS*). Az adatmodult használó űrlapok csak akkor érik el a rajta található elemeket, ha egységüknek *Implementation* részébe beépítjük a *uses uDM* sort

[1] Az *auto-create* űrlapok létrehozásának sorrendiségét a *Project/Options* menüpont hatására megjelenő párbeszédablak *Forms* lapján az *Auto-create forms* listában az elemek vonszolásával cserélhetjük fel.

(lásd 7.8. ábra). Ezt megtehetjük begépeléssel, vagy a *File/Use Unit* menüpont segítségével. Ha ezek után az adatmodulon elhelyezünk például egy *tblAnimals:TTable* komponenst, akkor a táblát használó űrlapon *DM.tblAnimals*-ként fogunk rá hivatkozni, hiszen a *tblAnimals* elem most már az adatmodul objektum (*DM*) egy nyilvános mezője.

7.7 Feladat: Egy táblán alapuló böngésző

Készítsük el a 7.8. ábrán látható alkalmazást. A megjelenítendő adatállomány a Delphi mintaadatbázis része: a DBDEMOS álnév által mutatott elérési útvonalon helyezkedik el, típusa Paradox, neve ANIMALS.DB.

Megoldás (🖫7_ANIMALS\PANIMALS.DPR)

A 16 bites Delphivel rendelkező Olvasók értelemszerűen adatmodul nélkül készítsék el az alkalmazást.

Lépések:

- Hozzuk létre az adatmodult (*File/New Data Module*), nevezzük el *DM*-nek, majd mentsük le (uDM.PAS).

  Helyezzünk el rajta egy *TTable* és egy *TDataSource* komponenst (*Data Access* paletta), majd állítsuk be ezek jellemzőit az alábbi táblázatnak megfelelően (mindenhol, ahol lebomló lista áll rendelkezésünkre, használjuk ezt ki):

Komponens	Jellemző	Érték
tblAnimals: TTable	DatabaseName	DBDEMOS *{ez a Delphihez tartozó mintaadatbázis álneve}*
	TableName	ANIMALS.DB
	Active	True *{ezzel nyitjuk meg a táblát}*
dsrAnimals:TDataSource	DataSet	tblAnimals

  Ezzel az adatelérés biztosítva van, jöhet a felhasználói felület.

- Tervezzük meg az alkalmazás űrlapját (*frmAnimals*). Még mielőtt bármit is elhelyeznénk rajta, „biztosítsuk az utat" az adatmodul felé:

```
unit uAnimals;
interface
uses  SysUtils, Windows...;
type
  TfrmAnimals = class(TForm)
     ...
implementation
{$R *.DFM}
uses UDM; {Itt, az Implementation részben hivatkozunk az adatmodul
          egységére}
...
```

Helyezzük el űrlapunkon az alábbi adatmegjelenítési komponenseket (*DataControls* paletta), majd állítsuk be a jellemzőiket a következő táblázat alapján[1]. S mivel a táblát már megnyitottuk, még tervezési időben látni fogjuk bennük az adatállomány első rekordjának mezőértékeit.

Komponens	Jellemző	Érték
eName: TDBEdit	DataSource	dm.dsrAnimals
	DataField	Name
eSize: TDBEdit	DataSource	dm.dsrAnimals
	DataField	Size
eWeight: TDBEdit	DataSource	dm.dsrAnimals
	DataField	Weight
eArea: TDBEdit	DataSource	dm.dsrAnimals
	DataField	Area
iBMP: TDBImage	DataSource	dm.dsrAnimals
	DataField	BMP
dbNavigator: TDBNavigator	DataSource	dm.dsrAnimals

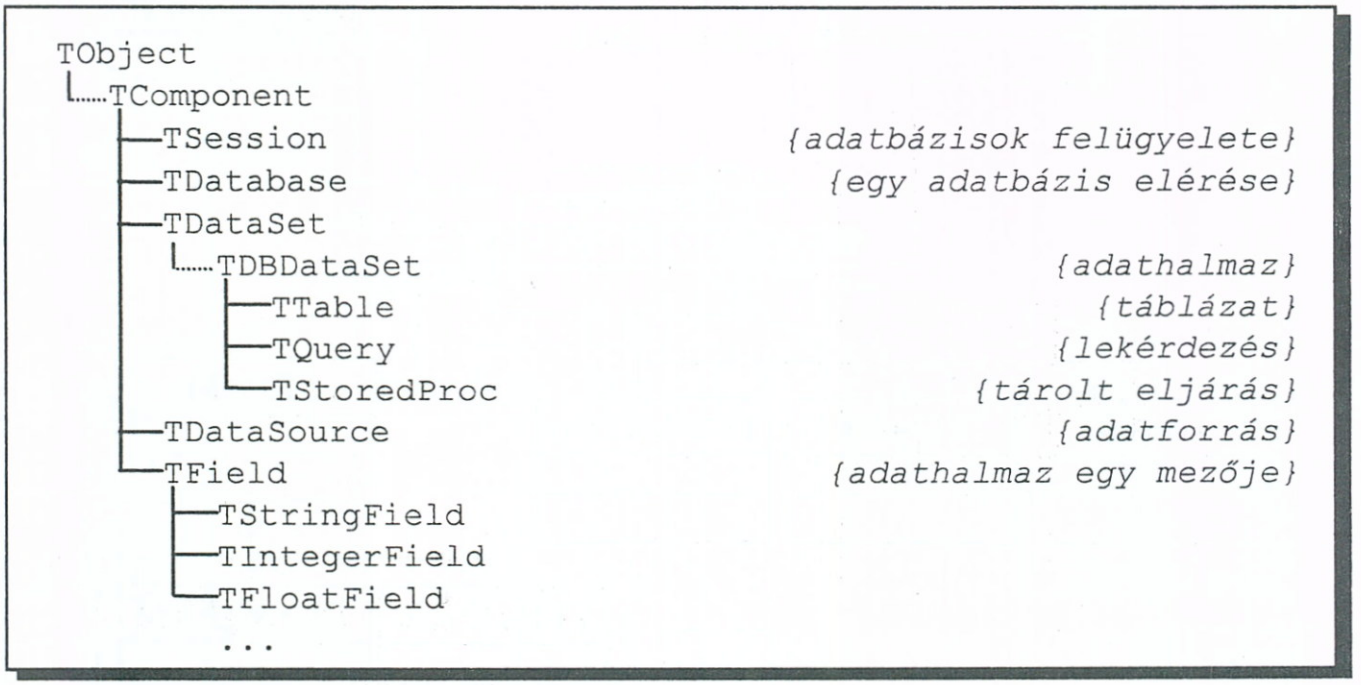
Indítsa el az alkalmazást! Próbálja ki a navigátorsor minden gombját.

Komolyabb alkalmazások elkészítése előtt közelebbről is meg kell ismerkednünk az adatbázis-kezelési komponensekkel. Ezzel foglalkozik a következő két fejezet.

[1] A *Panel*-ek, címkék és a kilépési gomb elhelyezését itt már nem tárgyaljuk.

8. Adatelérési komponensek

Ebben a hosszú, de remélem, ugyanakkor tanulságos fejezetben a Delphi adatelérési komponenseinek használatával ismerkedhetünk meg (8.1. ábra). Bemutatjuk az adathalmazok kezelésének módszereit – nem felejtkezve meg ezek hatékonyságáról sem –, az adathalmazok mezőinek kezelését, a származtatott mezők létrehozását, a fő-segéd űrlapok készítését, és még sok más – a Delphi alkalmazásainkhoz szükséges – technikát.



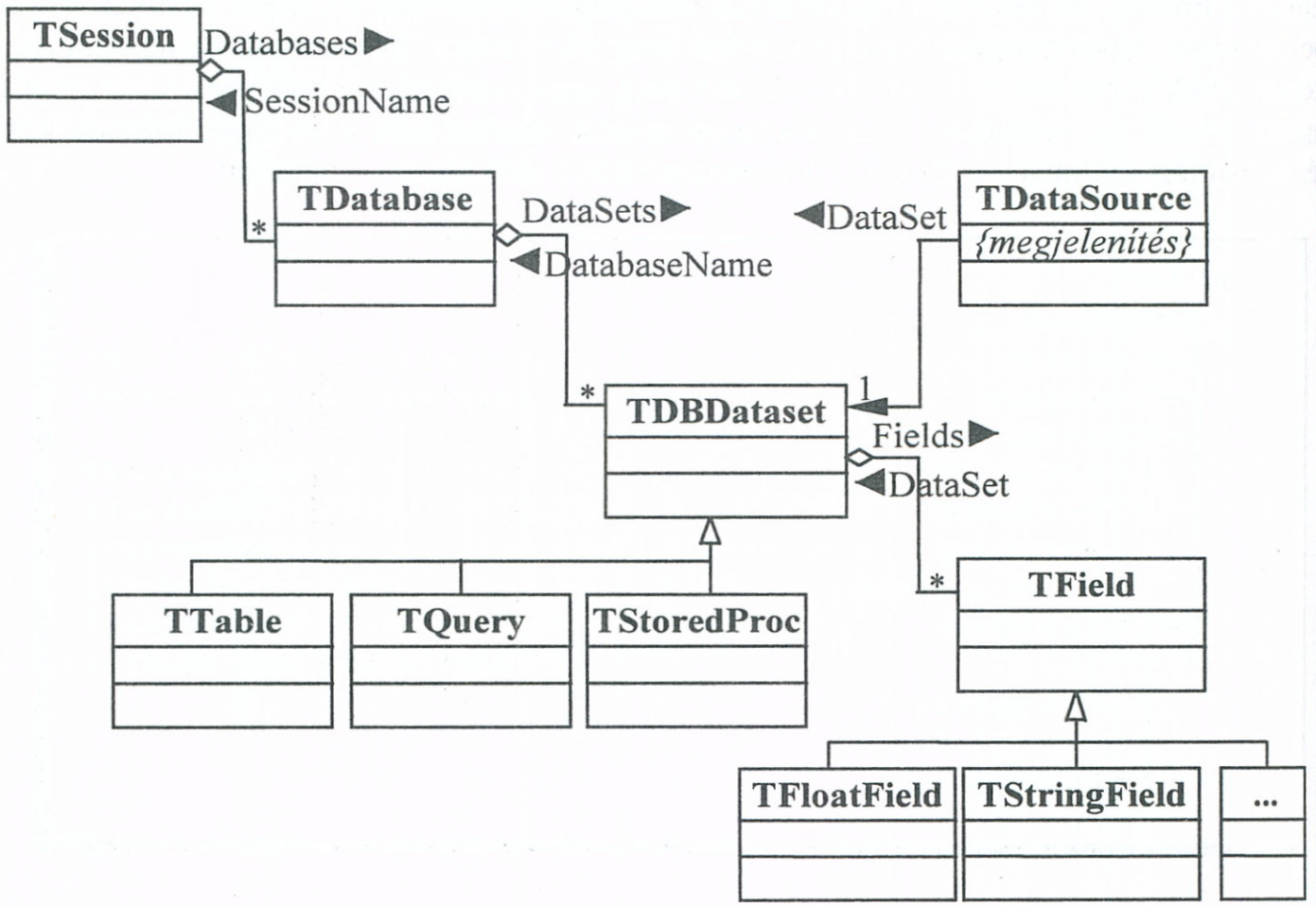
8.1. ábra. Adatelérési komponensek hierarchiája

8.1 Az adatelérési komponensek áttekintése

A Delphiben bevezetett adatelérési komponensek a valós adatbázisok szerkezetét mintázzák meg. A kezelt adatbázisok relációsak, mi mégis Delphiből objektumorientált szemlélettel közelítjük meg ezeket. A különböző komponensek kapcsolatait a 8.2. ábra mutatja be. Elemezzük együtt az ábrát!

Egy alkalmazásban legtöbbször egy adatbázissal dolgozunk, annak adathalmazait kezeljük. Ilyenkor az adatbázis elérését egy *TDatabase* komponens, adathalmazainak kezelését pedig a különböző *TTable, TQuery, TStoredProc* komponensek biztosítják. Mindhárom említett komponens egy adathalmaz adatainak elérését biztosítja; a különbség közöttük az,

hogy adataik különböző forrásokból származnak: a *TTable* komponens adatai egy fizikai adattáblából, a *TQuery* tartalma egy SQL lekérdezés lefuttatásából, a *TStoredProc* adatai pedig egy tárolt eljárás (bővebben lásd 14. fejezetben) végrehajtásából erednek. E három komponens közös vonásai Delphiben a *TDBDataset* közös ősben találhatók. Egy adatbázis-komponens tehát általában több adathalmazzal (*TDBDataSet*) áll kapcsolatban[1]; ezt a kapcsolatot az adatbázis *DataSets* jellemzője implementálja. Egy adathalmaznak általában több mezője is lehet (*Fields*). Delphiben az adathalmazok mezőinek egy-egy *TField* osztálybeli objektum felel meg, pontosabban a mező típusától függően vagy egy *TStringField*, vagy egy *TIntegerField*... objektum.



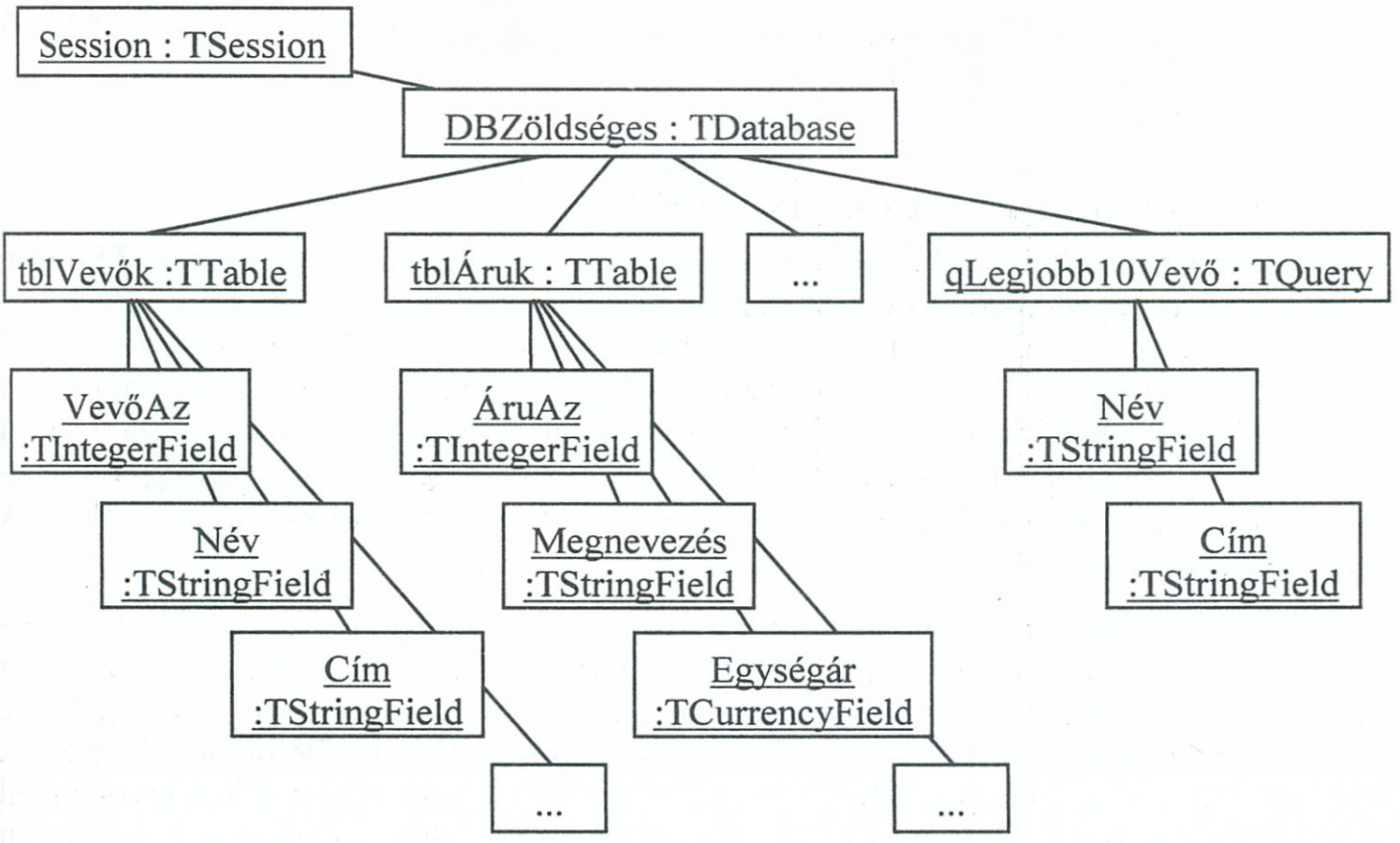
8.2. ábra. Adatelérési komponensek és kapcsolataik

Vannak olyan feladatok is, melyekben több *TDatabase* komponensre is szükségünk van: például akkor, ha a logikailag egy adatbázisba tartozó adataink fizikailag külön meghaj-

[1] Az UML jelölésben a * jelzi az egy-a-többhöz kapcsolatot. Ha a kapcsolatot jelző vonalon nincsenek nyilak, akkor a kapcsolat kétirányú (a mi ábránkon minden kapcsolat ilyen a TDataSource→TDBDataset kapcsolaton kívül). A vonal mentén a kapcsolatot megvalósító mezőket (ún. szerepneveket) tüntetjük fel. Például egy adatbázis-komponensnél a *DataSets* mezője tartalmazza az adatbázis adathalmazait, az adathalmazok pedig a *DatabaseName* jellemzőjükön keresztül kapcsolódnak egy adatbázishoz.

tókon találhatók (például az archív adatokat máshova mentjük), vagy akkor, ha két adatbázis között éppen adatokat cserélünk, esetleg konvertálunk egyik formátumból egy másikba... A lényeg tehát az, hogy egy Delphi alkalmazásban akár több adatbázis-komponenssel is dolgozhatunk. Az alkalmazásban szereplő – pontosabban az alkalmazás egy szálán (*thread*) kezelt – adatbázis-komponensek felügyeletét egy *TSession* komponens végzi el (bővebben lásd a 8.2. pontban).
Ha egy adathalmaz tartalmát meg szeretnénk jeleníteni egy űrlapon, akkor egy *TDataSource* komponenst kell ráirányítanunk (a *DataSet* jellemzőjén keresztül). Egy *TDataSource* egyszerre csak egy *TDBDataSet*-el állhat kapcsolatban, ebből veszi adatait. Az adatmegjelenítési komponenseket majd a *TDataSource* komponensre fogjuk ráállítani, így ezek az adatforrás tartalmát fogják tükrözni.

Tekintsük át mindezeket egy konkrét példán keresztül (8.3. ábra): Vázoljuk fel a sarki zöldséges adatbázisának adatelérési komponenseit. Tárolnunk kell a vevőket, árucikkeket, szállítókat, megrendeléseket... Ha Pisti bácsi – a zöldségbolt tulajdonosa – meg szeretné ajándékozni legjobb 10 vevőjét, akkor programunknak ki kell válogatnia ezeket ⇨ egy *TQuery* komponens segítségével *SQL* lekérdezést készítünk.



8.3. ábra. A sarki zöldséges nyilvántartó programjának adatelérési objektumai[1]

[1] Az UML jelölésben az objektumokat aláhúzzuk: *Objektum:Osztálynév*.

Immár áttekintettük a Delphi adatelérési komponenseit. Azonban még mielőtt bárkit is elrettentene ezek sokasága, tisztázzunk valamit: nem minden alkalmazásban van mindegyikükre szükség. Használatuk a konkrét feladattól függ:

- Egy kisebb fájl-szerver architektúrájú (például Paradox táblákat kezelő) adatbázisos alkalmazásban csak *TTable*, *TQuery*, *TField* és *TDatasource* komponensekre van szükség (lásd a könyvnyilvántartó feladat megoldását a 10., 12. és 13. fejezetekben).
- Egy kliens-szerver architektúrájú (például egy Interbase adatbázist kezelő) alkalmazásban a fentiek mellett még *TDatabase*-re és esetleg *TStoredProc*-ra is szükség lehet (lásd a hallgatói nyilvántartást a 14. fejezetben).
- A több szálon futó adatbázisos alkalmazásokban már külön *TSession* komponensekre is szükség lehet (lásd a 20. fejezet feladatát).

A következő pontokban részletesen is megismerkedhetünk az adatelérési komponensekkel. Az adatbázisok területén kicsit bizonytalanabb Olvasóknak azt javaslom, hogy ugorják át a következő két pontot (a *Session* és *TDatabase* komponensek bemutatását), hiszen mint az előbbiekben is láttuk, az egyszerűbb alkalmazásokban ezekre nincs is szükség. Az adathalmazok kezelését minden alkalmazásban használjuk, ezek bemutatása a 8.4. pontban kezdődik.

8.2 A TSession komponens

Minden adatbázisos alkalmazás indításakor automatikusan létrejön egy **Session: TSession** objektum. Szerepe az alkalmazás adatbázis-kapcsolatainak felügyelete. Ennek megfelelően jellemzői, metódusai nem egy adott adatbázisra, hanem az egész „munkafolyamatra" vonatkoznak. Például a *Databases* jellemzőjével lekérdezhetjük az aktuális adatbázisokat, a *GetAliasNames* metódusa az adatbázis-motor által felismert álneveket adja meg, a *GetDriverNames* pedig a különböző adatbázis-meghajtókat. A *Session* segítségével létrehozhatunk új álneveket (az *AddStandardAlias* és *AddAlias* metódusokkal), de törölhetünk is már létezőket (*DeleteAlias*)...

Álnevek létrehozására leginkább egy Delphi alkalmazás telepítésekor van szükség, olyankor kell az alkalmazásban használt álneveket a célgépen létrehoznunk és ráirányítanunk az adatok elérési útvonalára. Ezt a feladatot a 32 bites környezetekben már ellátja az InstallShield Express nevű Delphivel érkező segédprogram. Ezzel a Delphi alkalmazásunk számára telepítőprogramot készíthetünk: elég megmondanunk milyen állományokból áll az alkalmazás, milyen adatokon dolgozik, milyen álnevekre hivatkozik, majd a megadott információk alapján legenerálódik a telepítőprogram. Ez a telepítés végén létre fogja hozni az általunk kért álneveket is (bővebben lásd a 17. fejezetben).

A 16 bites Delphi alkalmazásokhoz saját kezűleg kell telepítőprogramot gyártanunk, itt tehát szükség lenne a *Session.AddAlias*-ra, viszont ebben a verzióban a *TSession* osztályban még nincs ilyen metódus. Az álneveket itt vagy közvetlen *BDE* hívásokkal, vagy álnevek kezelésére kidolgozott komponensekkel (például *AliasMan*) kell létrehoznunk.

Az automatikusan létrejövő *Session* komponensen kívül további *TSession* komponensekre a következő esetekben van csak szükség:

- Ha alkalmazásunk különböző hálózati gépeken található Paradox táblákat kezel. (Ez azért van, mert a *Session* tartja karban a Paradox táblák zárolási információira bevezetett PDOXUSRS.NET állományt; ennek az állománynak mindig az adatbázist tartalmazó gépen kell lennie (általában a gyökérkönyvtárban), hiszen a zárolási információk csak így elérhetők el több kliens gépről is. Egy *Session* komponens tehát csak egy hálózati gép adatbázisait képes ellátni.)

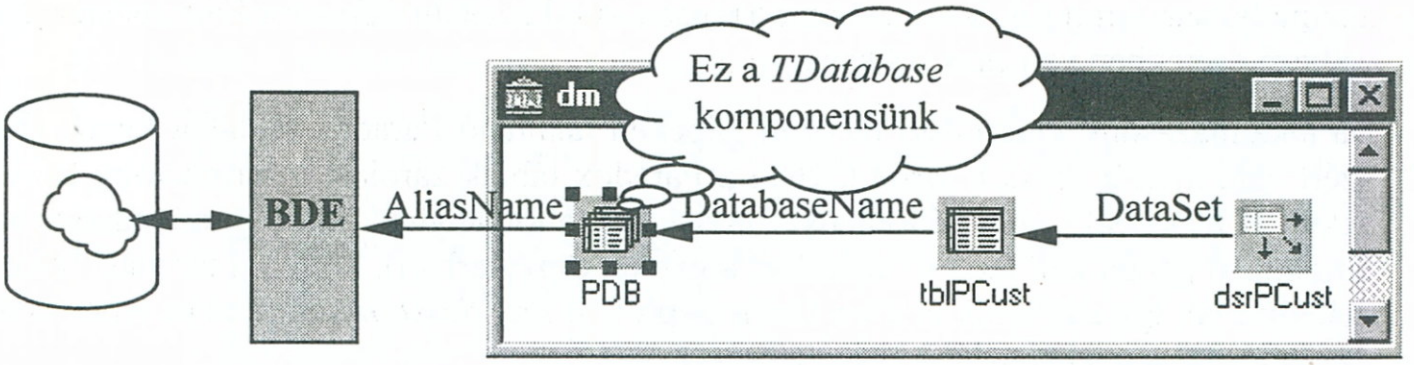
- Ha alkalmazásunk több szálon egy közös adatbázist kezel (itt már a kliens/szerver architektúrára kell gondolnunk). A többszálú (*multi-thread*) adatbázisos alkalmazásokban minden szálnak saját *Session* komponenssel kell rendelkeznie (több szálon, több munkafolyamat keretén belül, párhuzamosan futtathatunk lekérdezéseket azonos adatbázisból). Ilyenkor a *Session* komponenseket – az első kivételével – nekünk kell létrehoznunk akár tervezéskor (újabb *Session* objektumok elhelyezésével), akár futási időben (a *Sessions.OpenSession* metódussal). Bővebben lásd a 20. fejezetben, ahol egy két-szálon futó, Interbase adatbázist lekérdező alkalmazást készítünk.

8.3 A TDatabase komponens

- Helye az osztályhierarchiában: *TObject/TComponent/TDatabase*
- Szerepe: a *TDatabase* komponens egy konkrét adatbázis elérését biztosítja. Ha *Paradox* táblákkal dolgozunk, akkor nem kötelező a *Database* objektum explicit használata. Ilyenkor a *TTable*, *TQuery* komponensek *DatabaseName* jellemzőjükkel nem egy adatbázis-komponensre, hanem az álnévre hivatkoznak (lásd 8.2. ábra). Tervezési időben nem kell egyetlen *TDatabase* példányt sem elhelyeznünk az adatmodulon, ezt az alkalmazásunk indításakor automatikusan megteszi a rendszer.

  Az adatbázis-szerverekhez való csatlakozáshoz szükség van egy *Database* objektumra: ez biztosítja az utat az adatbázis felé, csatlakozáskor ez felelős a jelszó bekéréséért, metódusaival tranzakció-kezelést tudunk lebonyolítani.

  Ha *TDatabase* komponenst használunk, akkor a tábla-, lekérdezés- stb. komponensek a *DatabaseName* jellemzőjükkel nem az álnévre fognak hivatkozni, hanem az adatbázis-komponensre, az adatbázis-komponens pedig az *AliasName* jellemzőjével rámutat majd az adatok álnevére (lásd 8.4. ábra).



8.4. ábra. A *TDatabase* bekapcsolódása az adatelérési komponensek láncolatába

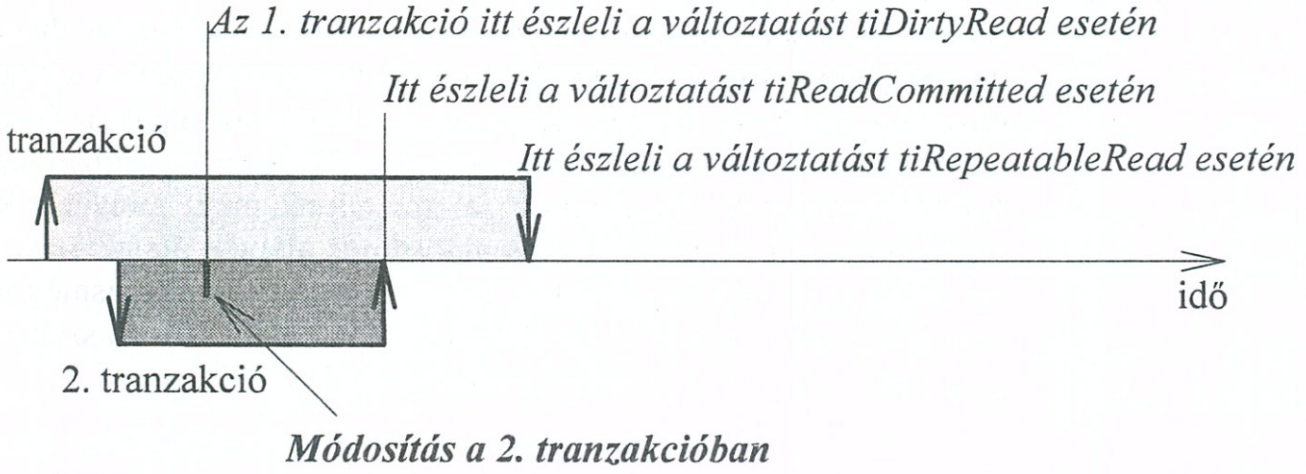
- Fontosabb jellemzői:
  - **AliasName**: az álnév, ami mögött adatbázisunk „megbújik"
  - **DatabaseName**: az adatbázis neve. Erre fognak majd a *TTable*, *TQuery*... komponensek hivatkozni.
  - **Connected**: Igazra állításával megtörténik az adatbázishoz való csatlakozás. Ilyenkor egy párbeszédablakban meg kell adnunk a felhasználó nevét és a jelszavát is.
  - **Params**: az adatbázishoz történő csatlakozás paraméterei. Az alkalmazás tervezése közben, annak érdekében, hogy nem kelljen állandóan, minden csatlakozásnál újból megadnunk az adatokat, tegyük a következőket: írjuk be a *Params* jellemzőbe a felhasználó nevét és jelszavát, majd állítsuk hamisra az adatbázis ***LoginPrompt*** jellemzőjét. Ezek hatására az adatbázishoz való csatlakozáskor a rendszer innen fogja kiolvasni az adatokat, így már a bejelentkezési párbeszédablakot sem jeleníti meg. Például ezt írhatnánk a *Params* jellemzőbe:

```
USER NAME=SYSDBA
PASSWORD=masterkey
```

Figyelem, ezt a technikát éles futáskor ne alkalmazzuk.

  - **TransIsolation**: ebben a jellemzőben a tranzakciók izolációs szintjét állíthatjuk be, vagyis azt, hogy a párhuzamosan futó tranzakciók mit és mennyit láthatnak egymás módosításaiból. Természetesen értéke a konkrét adatbázis-szerver által támogatott izolációs szintektől is függ. Lehetséges értékei: *tiDirtyRead* („piszkos olvasás", azaz a különböző tranzakciók látják egymás változtatásait még a tranzakciók befejezése előtt), *tiReadCommitted* (egy tranzakció az egyéb párhuzamosan futó tranzakciók által véglegesített („kommitált") módosításokat észleli csak) és *tiRepeatableRead* (legmagasabb izolációs szint: egy tranzakció elkezdésekor kiolvassa az adatokat, és mindvégig ezeken az értékeken dolgozik, mit sem

észlelve a párhuzamosan futó tranzakciók esetleges változtatásaiból) (lásd 8.5. ábra). Az alapértelmezett érték a *tiReadCommitted*.



8.5. ábra. Mikor észleli az 1. tranzakció a 2. által végrehajtott módosítást?

- Fontosabb metódusai:
  - **Open**: megnyitja az adatbázist
  - **Close**: bezárja az adatbázist
  - **StartTransaction**: tranzakció elindítása
  - **Commit**: tranzakció véglegesítése. Ilyenkor a tranzakció által végzett módosítások lementődnek.
  - **Rollback**: tranzakció visszagörgetése. A tranzakció által végzett módosítások elvesznek, visszaáll a tranzakció elindításakor érvényes konzisztens állapot.

A tranzakció-kezelés nem csak adatbázis-szerverek esetén használható, hanem például Paradox táblák esetén is igénybe vehető. Természetesen az igazi hatékonyság az adatbázis-szerverekkel érhető el.

Tanulmányozza át a *Paradox* és *InterBase* adattáblákon megvalósítható tranzakció-kezelést a 8_TRANZAKCIOK\PTRANZAKT.DPR alkalmazás segítségével! (Az *Interbase* adatbázis megnyitásakor a felhasználó = SYSDBA, a jelszó = masterkey).
Módosítson több rekordot, majd vonja vissza a *Rollback* gombbal. Utána újból módosítson, majd véglegesítse. Figyelje meg, hogy a véglegesített tranzakciókat már nem lehet visszagörgetni.

- Eseményjellemzője:
  - **OnLogin**: az adatbázishoz való csatlakozáskor következik be. Erre az eseményre beépíthetjük például egy „saját szánk íze szerinti" bejelentkezési ablak megjelenítését, ekkor a saját párbeszédablakunk fogja az adatokat bekérni és átadni az adatbázis-szervernek (lásd a 14. fejezet feladatában).

Az ***OnLogin*** eseményt használhatjuk egy sokkal furfangosabb célból is: általában az alkalmazás-logikát az adatbázisba igyekszünk beépíteni, azonban a gyakorlatban ennek kisebb-nagyobb hányada át szokott kerülni a kliens oldalra. Ilyenkor fontos biztosítanunk azt, hogy a felhasználók csak a kliens oldali alkalmazáson keresztül léphessenek be az adatbázisba, ne tudjanak esetleg más eszközzel is az adatok közelébe férkőzni, hiszen akkor megkerülnék az alkalmazás-logika kliens oldalra épített részét. Ezt a következőképpen valósíthatjuk meg: a felhasználóknak egy „hamis" jelszót adunk meg, melyet a kliens oldali programban egy bizonyos algoritmus alapján átalakítunk, majd ezt adjuk át az SQL-szervernek. Így a felhasználók által ismert jelszó nem lesz hatásos egyéb eszközökkel történő belépéskor (hiszen azok a meg nem változtatott jelszóval próbálnának belépni).

```
procedure TDM.DatabaseLogin(Database: TDatabase;
                            LoginParams: TStrings);
var jelszo:Char[8];
    felhasznalo:String[31];
begin
   {felhasználó és jelszó bekérése..., majd
    következhet a jelszó átalakítása}
   jelszo:= Atalakit (jelszo);
   With Loginparams Do
      begin
         {paraméterek előkészítése a LoginParams-ba,
         így adjuk ezeket át az SQL-szervernek.}
         Clear;
         Add('USER NAME='+ felhasznalo);
         Add('PASSWORD='+ jelszo);
      end;
end;
```

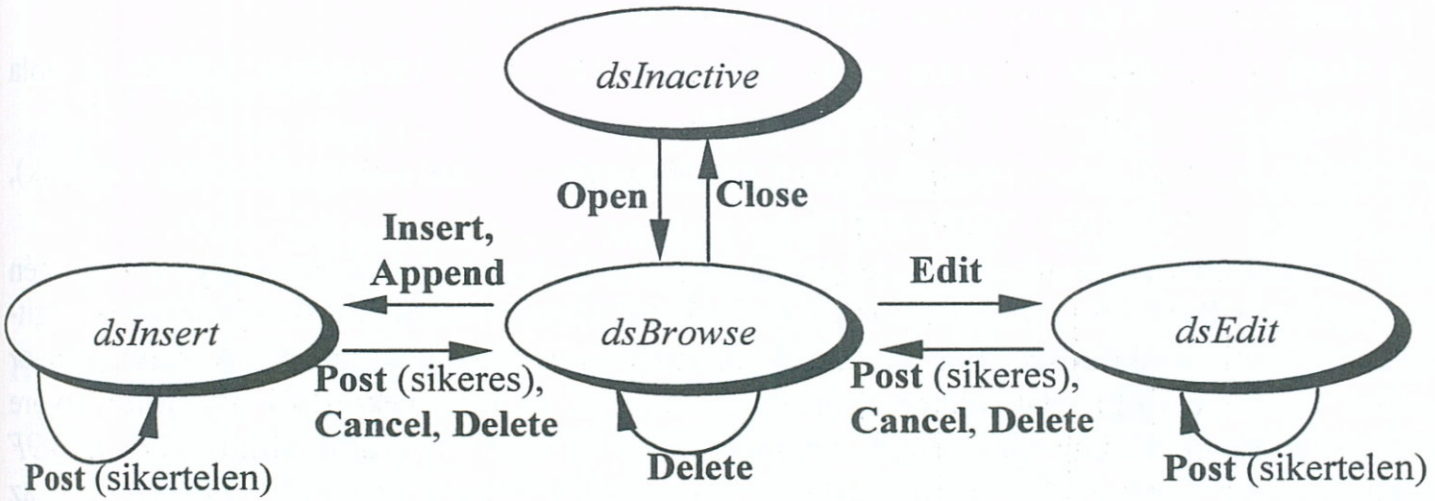
8.4 Az adathalmazok kezelése: TDBDataSet osztály

Az itt bemutatottak egyformán vonatkoznak a *TTable*, *TQuery* és *TStoredProc* komponensekre, hiszen ezek mind különböző forrásokból származó adathalmazokra vonatkoznak. Az elsőnél az adathalmaz egy fizikai táblából származik, a másodiknál egy lekérdezés eredményeként, a harmadiknál pedig egy tárolt eljárás végrehajtásaként.

8.4.1 Adathalmazok állapotai

Az adathalmazok lehetséges állapotait a 8.6. ábra mutatja. Egy adott pillanatban az állapotát a **State** jellemzője tartalmazza. Ez vagy *dsInactive*, vagy *dsBrowse*, vagy *dsEdit*...
Az adathalmaz megnyitása előtt inaktív (***dsInactive***) állapotban van. Ilyenkor még tartalmához nem férünk hozzá, még olvasásra sem. Megnyitásakor (az ***Open*** metódus vagy az ***Active := True*** utasítással) átkerül ***dsBrowse*** („böngészés") állapotba.

Ha módosítani szeretnénk egy rekordot, akkor az ***Edit*** metódussal át kell „billentenünk" ***dsEdit*** állapotba, ott elvégezzük a módosításokat, majd ***Post***-al mentjük ezeket. Természetesen egy mentés lehet sikeres vagy sikertelen. Sikertelen akkor, ha például nem megengedett értéket állítottunk be, vagy ha a mezőértékek egyediségét nem tartjuk be (kulcsok vagy egyedi indexek esetén)... Ha tehát a mentés sikertelennek bizonyul, akkor az adathalmaz mindaddig ***dsEdit*** állapotban marad, amíg vagy visszavonjuk a változtatásokat (***Cancel*** metódus), vagy kijavítjuk a hibás adatot, és úgy mentjük le (még egy *Post*). Mindkét esetben visszakerül ***dsBrowse*** állapotba.
Az új rekordok felvitele hasonló módon történik. Ekkor az ***Insert*** vagy ***Append*** metódussal egy új, üres rekordot veszünk fel, az adathalmaz ezzel ***dsInsert*** állapotba jut. Beállítjuk az új értékeket, majd következik a mentés vagy a visszavonás.
Egy adathalmaz aktuális rekordját *dsBrowse*, *dsInsert* és *dsEdit* állapotból is ki lehet törölni. Mindhárom esetben ezt a ***Delete*** metódussal valósítjuk meg; a törlés után adathalmazunk újból *dsBrowse* állapotba kerül.



8.6. ábra. Adathalmazok leegyszerűsített állapotdiagramja[1]

8.4.2 Adathalmazok nyitása, zárása

Az adathalmazokat tervezési és futási időben is meg lehet nyitni, és be lehet zárni. Tervezéskor csak az *Active* jellemzőjük segítségével tehetjük ezt, míg futáskor használhatjuk az *Active* jellemzőt, valamint az *Open* és *Close* metódusokat is.

[1] Egy adathalmaz további állapotokban is lehet, ezek számunkra talán kevésbé fontosak. Ezek olyan ideiglenes állapotok, melyekben az adathalmaz csak nagyon rövid ideig tartózkodik. Például kereséskor az adathalmaz *dsSetKey* állapotba kerül, szűréskor (*Filter* beállításakor) *dsFilter* állapotba... Az ábra a Borland Delphi for Windows 95 & Windows NT: Database Application Developer's Guide című könyvben található ábra alapján készült.

	Tervezési időben	Futási időben
Nyitás	Active := True	Open *{vagy}* Active := True
Zárás	Active := False	Close *{vagy}* Active := False

8.4.3 Mozgás az adathalmazban

Minden nyitott adathalmazban létezik egy rekordmutató (továbbiakban mutató), ami az aktuális rekordon áll. Ennek mezőértékeire vonatkozik a szerkesztés és törlés, valamint ezek értékei jelennek meg a különböző megjelenítési komponensekben is (*DBEdit, DBListbox...*). Az aktuális rekord megváltoztatását az ún. navigációs metódusokkal érhetjük el. Ezek a következők:

- **First**: a mutatót az adathalmaz első rekordjára helyezi
- **Last**: a mutatót az adathalmaz utolsó rekordjára helyezi
- **Next**: a következő rekordra ugrik. Ha nincs következő (mert már elértük a tábla végét), akkor a mutató a régi helyén marad.
- **Prior**: az előző rekordra ugrik. Ha nincs előző (eleve a tábla első rekordján vagyunk), akkor a mutató a régi helyén marad.

Egy adathalmazra vonatkozólag le tudjuk kérdezni, hogy az elején vagy a legvégén vagyunk-e, a **BOF** (***Beginning-Of-File***) és az **EOF** (***End-Of-File***) logikai jellemzők segítségével. Ezek működését már az előző feladatban is tapasztalhattuk. A navigátorsor „*előző*" gombja (◄) csak akkor szürkül el, miután az első rekordról az előtte levőre próbálunk lépni. Ekkor tapasztalja ugyanis, hogy az első rekordon állunk, így a *BOF* jellemző igaz értéket kap. Természetesen ugyanezt a „késett reakciót" tapasztaljuk az utolsó rekord esetén is.

Első látásra ez talán kicsit bosszantónak, érthetetlennek tűnik, azonban még ne vonjunk le elsietett következtetéseket. Egy adathalmazt több program, hálózatos környezetben pedig több gép is használhat. Amikor például az utolsó előtti rekordról *Next*-tel a következőre állunk (valamilyen feldolgozás céljából), a BDE nem lehet biztos abban, hogy a feldolgozás után ez a rekord még mindig az utolsó lesz; lehet, hogy azóta a tábla más felhasználói új rekordokat helyeztek el az addigi utolsó után.

Az, hogy egy adathalmaz első rekordján állunk, csak a következő esetekben biztos:

- Ha pont most nyitottuk meg az adathalmazt
- Ha most hívtuk meg a *First* metódust
- Ha egy *Prior* metódus sikertelennek bizonyult, jelezvén, hogy nincsenek előző rekordok.

Az, hogy egy adathalmaz utolsó rekordján állunk, csak a következő esetekben biztos:

- Ha most nyitottunk meg egy üres adathalmazt

- Ha most hívtuk meg a *Last* metódust
- Ha egy *Next* metódus sikertelennek bizonyult, jelezvén, hogy nincs több rekord.

Adathalmazok ciklikus feldolgozása

Feldolgozás előre

```
With Table1 Do
  begin
    First;
    While Not EOF Do
      begin
        {akt. rekord feldolgozása}
        Next;
      end;
  end;
```

... és vissza

```
With Table1 Do
  begin
    Last;
    While Not BOF Do
      begin
        {akt. rekord feldolgozása}
        Prior;
      end;
  end;
```

Figyelem!

- Az adathalmazok feldolgozásánál kerüljük a hátultesztelő ciklust. Üres adathalmaz esetén egyetlen rekord feldolgozásának sem szabad lefutnia.
- A ciklusmagból sose felejtsük ki a *Next* vagy a *Prior* metódus hívását. Könnyen végtelen ciklusban találhatjuk magunkat.

Hatékonysági megfontolások

Egy adathalmaznak valamilyen célból történő programból való végigolvasása jelentősen lassúbb, mint az ugyanazt a feladatot ellátó SQL lekérdezés. Főleg a nagyobb terjedelmű adathalmazoknál figyeljünk erre.

Számoljuk például össze az alkalmazottak táblában (*DM.tblEmployee*) a 200 ezer Ft-nál nagyobb fizetésűeket. Ezt megtehetjük végigszaladva a táblán programból, de megtehetjük egy egyszerű SQL lekérdezéssel is. **Az első megoldás kb. 1,1 másodpercet, míg a lekérdezés kb. 0,15 másodpercet vesz igénybe.** (A méréseket egy 10000 rekordot tartalmazó táblán végeztük, egy Pentium 100MHz, 32 MB RAM-os Windows NT operációs rendszerrel rendelkező gépen).

```
{8_NAVIGACIO\PNAVIG.DPR}
{Első megoldás: a táblán programból szaladunk végig}
procedure TfrmNavig.Gazdagok1Click(Sender: TObject);
var  Szamlalo:Integer;
begin
  With DM.tblEmployee Do
    begin
      Szamlalo:=0;
      First;
```

```
        While Not EOF Do
           begin
              {Így hivatkozunk egy mező értékére (lásd később)}
              if FieldByName('Salary').Value>=200000 Then
                 Inc (Szamlalo);
              Next;
           end;
     end;
   ShowMessage(Format('Gazdagok száma: %d',[Szamlalo]));
end;

{Második megoldás: egy SQL lekérdezés segítségével számoljuk össze a
gazdagokat. Itt csak az SQL parancsot tekintjük át, a lekérdezések
Delphiből történő futtatásával később foglalkozunk.}
SELECT COUNT(Empno)
FROM MEmploy
WHERE (Salary > 200000)
```

Ha a lassúság ellenére mégis úgy döntenénk, hogy az adathalmaz végigolvasásával kívánunk elvégezni egy feldolgozást, akkor ne felejtsük el kikapcsolni az adatmegjelenítést a feldolgozás idejére. Ha bekapcsolva hagynánk a megjelenítést, akkor az összes – ebből az adathalmazból „táplálkozó" – adatmegjelenítési komponens értékét a rendszernek frissítenie kellene minden egyes rekordnál, ez pedig nagyon megnövelné a feldolgozás teljes időtartamát. Példánkban, a nagy fizetésűek kiválogatása **megjelenítéssel együtt 54 másodperc**ig tartott, míg **megjelenítés nélkül csupán 1,1 másodperc**ig.

Az adatmegjelenítés kikapcsolását az adathalmaz **DisableControls** metódusával, a bekapcsolást pedig az **EnableControls** metódusával érhetjük el. Figyelem, ha a feldolgozás közben valamilyen hiba következik be, akkor a megjelenítést még mindenképpen vissza kell állítanunk. Erre használjuk a *Try...Finally* utasítást.

Az előző feladat „biztonságos" változata:

```
procedure TfrmNavig.Gazdagok1Click(Sender: TObject);
var  Szamlalo:Integer;
begin
   With DM.tblEmployee Do
      begin
         DisableControls;
         Try
            Szamlalo:=0;
            First;
            While Not EOF Do
               begin
                  If FieldByName('Salary').Value>=200000 Then
                     Inc (Szamlalo);
                  Next;
```

```
            end;
          ShowMessage(Format('Gazdagok száma: %d',[Szamlalo]));
        Finally
          EnableControls;
        End;
      end;
end;
```

Végezze el a különböző hatékonysági teszteket a saját gépén is a 8_NAVIGACIO\PNAVIG.DPR alkalmazás segítségével. Az alkalmazás létrehoz egy 10000 rekordos táblázatot, majd különböző feldolgozásokat végez rajta programból és lekérdezésekkel. A táblázat programból való létrehozásának, valamint a lekérdezések futtatásának módját később ismertetjük.

Ugyanezt a feladatot a 20. fejezetben is megoldjuk, ott viszont a számlálási algoritmusokat két párhuzamos szálon (*thread*) futtatjuk le.

8.4.4 Rekordok szerkesztése, törlése, új rekordok felvitele

Rekordok szerkesztése

Az adathalmazban mindig az aktuális rekordot módosíthatjuk, azt is csak akkor, ha az adathalmaz szerkeszthető (nem *read-only*), és ráadásul még *dsEdit* állapotban is van. Valójában az, hogy egy adathalmaz írható-e vagy csak olvasható, több tényezőtől is függ: például attól, hogy tervezési időben csak olvashatóvá tettük-e vagy sem; vagy attól, hogy más felhasználók nem zárolták-e előlünk. A lekérdezéseknél az is egy fontos szempont, hogy egy vagy több táblán alapulnak-e, tartalmaznak-e aggregáló függvényeket (*SUM, COUNT* stb.)... A rendszer kiértékeli a szerkeszthetőséget, és ezt a **CanModify** jellemzőből mi is megtudhatjuk.
Egy rekord mezőértékeinek átírása előtt meg kell győződnünk arról is, hogy a tábla *dsEdit* állapotban van-e. Ha még nincs, akkor az *Edit* metódussal „átbillentjük" ebbe az állapotba.

A szerkesztés általános formája a következő:

```
With Table1 Do
  If CanModify Then
    Begin
      If Not (State in [dsEdit, dsInsert]) Then
        Edit;
      FieldByName('Mező1név').Value := érték1;
      FieldByName('Mező2név').Value := érték2; ...
      Post;
    end
```

Példánkban a módosítások mentése a *Post* metódus hatására következik be („postázás"). Ha viszont nem hívjuk meg explicit módon a *Post* metódust, akkor ez még nem jelenti azt, hogy a módosítások elvesznek. Akárcsak más rendszerekben, Delphiben is automatikusan lementődnek a változtatások egy másik rekordra való átlépéskor.

Ha a módosításokat vissza szeretnénk vonni, akkor ezt még postázás előtt megtehetjük a *Cancel* metódus segítségével.

```
With tblAlkalmazottak Do
 begin
  Edit;
  FieldByName('Fizetés').Value := FieldByName('Fizetés').Value * 2;...
  Cancel;
 end
```

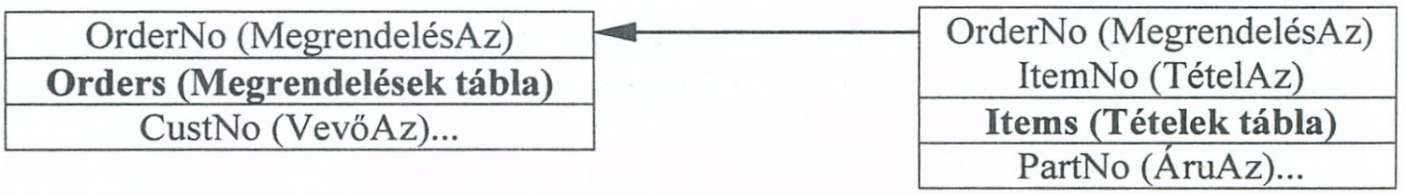
Ha már lementettünk (postáztunk) egy módosítást, akkor ezt már csak abban az esetben vonhatjuk vissza, ha a módosító művelet egy még nem véglegesített tranzakciónak része. Ekkor az adatbázis-komponens *RollBack* metódusa visszaállítja a tranzakció elkezdésének pillanatában tapasztalt állapotot.

Rekordok törlése

A *Delete* metódus letörli az adathalmaz aktuális rekordját.

```
Table1.Delete;
```

Természetesen vannak esetek, amikor ez sikertelen lesz. Például – a hivatkozási integritás szabályai alapján – az egyes oldali táblák rekordjait mindaddig nem lehet kitörölni, amíg a többös oldalon tartoznak hozzájuk értékek. (Egy megrendelést nem törölhetünk ki, ha vannak tételei).
Feladatainkban a Delphi mintaadatbázisát használjuk, hiszen ez már készen van, és adatokkal fel van töltve. Emiatt sajnos angol mezőnevekkel kell dolgoznunk. Az angolul kevésbé tudó Olvasókra gondolva zárójelben feltüntettük a mezők magyar megfelelőjét is.



8.7. ábra. Egy megrendeléshez több tétel tartozik[1]

Ha programunkban nem kezeljük le ezt az esetet, akkor a következő angol üzenet fog megjelenni: „*Master has detail records*" („*az egy-oldali rekordhoz a több-oldalon értékek tartoznak*"). Ezzel az üzenettel legtöbbször a program végfelhasználói nem tudnak mit

[1] Az adatmodellek elkészítésénél a következő ábrázolási technikát használjuk: egy táblának (egyedtípusnak) egy három részes doboz felel meg: a felső részében az elsődleges kulcsot alkotó mezőt (-ket), középen a tábla nevét, alul pedig a többi mezőt tüntetjük föl. A kapcsolatot jelző vonalat a kapcsolódó mezők közé húzzuk, a nyíl mindig az 1-es részre mutat (ugyancsak a kapcsolat fokának jelzésére rajzolhatnánk „csirkelábat" is, annak mindig a többös részen kellene lennie).

kezdeni. A mi feladatunk tehát egy barátságosabb üzenet megjelenítése az ilyen esetekben. Ezt többféleképen is megtehetjük:

1. **Első megoldás:** ha a törlést egy saját gombunkra építjük be (nem a navigátorsor törlőgombját használjuk), akkor a *Törlés* gombunk kattintására a következőket írjuk:

```
procedure TfrmMasterDetail.btnTorlesClick(Sender: TObject);
begin
  Try
     DM.tblOrders.Delete;
  Except
     On EDatabaseError Do
        begin
           ShowMessage('A megrendelés nem törölhető, mivel'+
                       ' vannak tételei');
           Abort;
        end;
  End;
end;
```

2. **Második megoldás:** ha több helyről is lehet a törlést kezdeményezni (például a navigátorsor gombjával, a megrendelés rácsból billentyűzetről...), akkor legkényelmesebb (és elegánsabb is) ha a *Megrendelés* tábla *BeforeDelete* eseményébe építjük be az ellenőrzést. Ha itt azt tapasztaljuk, hogy a törlés nem lehetséges, akkor az *Abort* metódussal megszakítjuk a már beindult törlési folyamatot.

```
procedure TDM.tblOrdersBeforeDelete(DataSet: TDataSet);
begin
  If tblItems.RecordCount >0 Then
     begin
        ShowMessage('A megrendelés nem törölhető, mivel vannak'+
                    ' tételei');
        Abort;
     end;
end;
```

3. **Harmadik megoldás:** az adathalmaz *OnDeleteError* eseményjellemzőjének segítségével. Implementációja bonyolultabb, lásd a 8.4.7. pontban.

Ez a példa csupán a technika bemutatására való. Remélhetőleg senki sem fog ennek alapján egy „éles" alkalmazásban létező megrendelést (számlát) „csak úgy" kitörölni.

Új rekord felvitele

Új rekordot az adathalmaz **Insert** vagy **Append** metódusaival vehetünk föl. Formájuk:

```
With Table1 Do
  begin
    Insert; {vagy Append}
    FieldByName('Mezőlnév').Value := érték1;
    FieldByName('Mező2név').Value := érték2;
    ...
    Post;
  end;
```

Postázás előtt szükség esetén még vissza lehet vonni az új rekord felvitelét a *Cancel* metódussal.

8.4.5 Keresés az adathalmazban

A keresés és szűrés fogalma

Keresésnek (Searching) nevezzük azt a folyamatot, amikor egy adathalmaz sok rekordja közül **egy** adott rekordot szeretnénk megtalálni (*például a sarki zöldséges árui közül rákeresünk a Paradicsomra*). Keresés után a talált rekorddal valamit kezdeni szeretnénk: lehet, hogy csak az adataira vagyunk kíváncsiak (*Mennyibe kerül 1 kg paradicsom?*). Az is lehet viszont, hogy át akarjuk írni a talált rekord adatait (*A paradicsom megdrágult ⇨ változik az egységára*); ez esetben a talált rekordot aktuálissá kell tennünk, hogy módosíthassuk mezőértékeit. A keresés semmiképpen sem eredményezi az adathalmaz leszűkítését. (*Adataink között továbbra is ott marad a banán, a narancs, a sárgarépa...*)

Egy adathalmaz valamilyen feltételnek megfelelő rekordjainak leválogatási folyamatát **szűrésnek (Filtering)** nevezzük (*például egy telefonkönyvből csak az 'A' kezdőbetűs előfizetők adataira vagyunk kíváncsiak*). Ezzel a folyamattal leszűkítjük az adathalmazt a feltételnek megfelelő rekordok halmazára. A szűréssel bővebben a következő pontban (8.4.6.) foglalkozunk.

A keresés megvalósítása

Delphiben a következő metódusokkal lehet adathalmazokban keresni (az első kettő csak a 32 bites Delphi verziókban érhető el, az utolsó kettő pedig csak *TTable* komponensekre alkalmazható)[1]:

- **Locate**: segítségével akár indexelt, akár nem indexelt mezők értékeire kereshetünk rá. (Ha van index, akkor azt a rendszer automatikusan felhasználja.) A kurzor az **első** feltételnek megfelelő rekordra kerül. A *Locate* függvény egy logikai értékkel tér vissza, mely jelzi a keresés sikerességét. (*Ezt használnánk, ha a paradicsom megtalálása*

[1] A különböző keresési módszereket kipróbálhatja a 8_KERESES\PSEARCH.DPR alkalmazásban.

után egységárát módosítani szeretnénk.) A *Locate* hívása előtt beállíthatunk bizonyos keresési opciókat is: azt, hogy különböztesse-e meg a kis és nagy betűket (*Paradicsom = PaRaDicsom*), valamint azt, hogy teljes vagy részleges keresést végezzen-e (*'Para' rákeresésére megtalálja a 'Paradicsom'-ot*). Természetesen az opciók csak a karakterlánc típusú mezőkre vonatkoznak.

Például:

```
{Keressük meg azt a vevőt, amelynél a Company = 'Blue Sports' (a
kis/nagy betűket nem különböztetjük meg):}
With DM.tblCust Do
  begin
     If Not Locate ('Company', 'Blue Sports',[loCaseInSensitive])
        Then ShowMessage('Nincs találat!');
  end;

{Keressük meg az USA-beli 'Blue Sports' nevű vevőt:}
With DM.tblCust Do
  begin
     If Not Locate ('Country;Company',
                    VarArrayOf(['US','Blue Sports']),
                    [loCaseInSensitive, loPartialKey]) Then
        ShowMessage('Nincs találat!');
  end;
```

Ha tehát több mező értékére szabunk keresési feltételt, akkor a mezőket ';'-vel választjuk el egymástól.

A mezők konkrét értékei elvileg akármilyen típusúak lehetnek. Emiatt, a *Locate* második paramétereként egy ***Variant*** (típusnélküli) paramétert vár. Ha egy mezőre keresünk csak, akkor ennek értékét simán átadhatjuk (lásd első példa), ha viszont több értéket kell átadnunk, akkor a ***VarArrayOf*** függvény segítségével fel kell építenünk egy *Variant* elemű tömböt (lásd második példa).

- **Lookup**: működésileg nagyon hasonlít a *Locate*-ra, a különbség a kettő között csak az eredményben van. Ez a metódus is rákeres az adott feltételnek megfelelő első rekordra, de azt nem aktualizálja, hanem mezőértékeit eredményként visszaadja. (*Ezt használnánk, ha csak a paradicsom egységára érdekelne.*) Keresési opciókat itt nem állíthatunk be.

  Például:

```
{Kérdezzük le a 'Blue Sports' vevőnk azonosítóját (CustNo):}
Var FoundRec: Variant;
...
With DM.tblCust Do
  begin
     FoundRec:= Lookup ('Company', 'Blue Sports', 'CustNo');
     If VarIsNull(FoundRec) Then
        ShowMessage('Nincs találat!')
```

```
    Else
      ShowMessage('Az azonosító: ' + String(FoundRec) )
  end;

{Kérdezzük le az USA-beli 'Blue Sports' nevű vevő azonosítóját:}
Var FoundRec: Variant;
...
With DM.tblCust Do
  begin
    FoundRec:= Lookup ('Country;Company',
                       VarArrayOf(['US','Blue Sports']),
                       'CustNo');
    If VarIsNull(FoundRec) Then
      ShowMessage('Nincs találat!')
    Else
      ShowMessage('Az azonosító: ' + String(FoundRec) )
  end;

{Kérdezzük le az USA-beli 'Blue Sports' nevű vevő azonosítóját,
városát és utolsó megrendelésének dátumát (LastInvoiceDate):}
Var  FoundRec: Variant;
    i:Integer; S:String;
...
With DM.tblCust Do
  begin
    FoundRec:= Lookup ('Country;Company',
                        VarArrayOf(['US','Blue Sports']),
                       'CustNo;City;LastInvoiceDate');
    If VarIsNull(FoundRec) Then
      ShowMessage('Nincs találat!')
    Else
      begin
        S:='';
        For i:= VarArrayLowBound(FoundRec,1) To
                VarArrayHighBound(FoundRec,1) Do
          S:= S+ String(FoundRec[i])+'; ';
        S:= Copy(S,1,Length(S)-2);
        ShowMessage(S);
      end;
end;
```

Ha több mező értékét kérdezzük le, akkor az eredmény nem egy sima *Variant*, hanem egy *Variant*-okból álló tömb. Általános esetben a *Variant* típusú tömbök több dimenzióval is rendelkezhetnek, ezt a ***VarArrayDimCount*** függvénnyel kérdezhetjük le. A *Lookup*-nál erre nincs szükség, hiszen ez mindig csak egyetlen rekord adataival tér vissza; az eredménytömb tehát egydimenziós. Az indexek határértékeit a ***VarArrayLowBound*** és ***VarArrayHighBound*** függvényekkel kérdezzük le. Két paramétert kell

ezeknek átadnunk: az első a *Variant* tömb, a második pedig annak a dimenziónak a sorszáma, amelyiknek az indexhatárait le akarjuk kérdezni (példánkban 1).

- **FindKey** (csak *TTable* komponenseknél használható): az aktuális index oszlopaiban keres rá egy adott rekordra. Pontos találat esetén a kurzort a megfelelő rekordra helyezi. Egy logikai értékkel tér vissza, mely jelzi a keresés sikerességét.

Például:

```
{Keressünk rá az 1231-es azonosítóval rendelkező vevőnkre; Az
aktuális indexelt mező a CustNo}
If Not DM.tblCust.FindKey([1231]) Then
  ShowMessage('Nincs találat!')
...

{Keressünk rá az USA-beli Blue Sports nevű vevőre; Az aktuális in-
dex Country, és azon belül Company szerint rendezett}
If Not DM.tblCust.FindKey(['US', 'Blue Sports']) Then
  ShowMessage('Nincs találat!')
```

Ennél a keresési módszernél nagyon fontos, hogy a tábla indexét még a keresés előtt beállítsuk. Ezt a tábla ***IndexFieldNames*** jellemzőjével fogjuk megvalósítani. Például:

```
DM.tblCustno.IndexFieldNames:= 'Custno';
{keresés Custno szerint}...
DM.tblCustno.IndexFieldNames:= 'Country;Company';
{keresés Country és azon belül Company szerint}...
```

A Paradox és dBase táblák esetén az ***IndexFieldNames*** jellemzőbe csak létező indexszel rendelkező mezőket állíthatunk be. Az SQL szerverek esetén viszont e jellemzőbe beírhatunk bármilyen mezőt (-ket), ezek alapján az SQL szerver felépít egy ideiglenes indexet, és ezt használja a keresés idejére.

De térjünk vissza a Paradox táblákhoz: **hogyan lehet megállapítani, hogy egy tábla rendelkezik-e az adott mező (-k) szerinti indexfájllal?** Elvileg minden tábla összes létező indexét programból le lehet kérdezni, ezzel később foglalkozunk. Most viszont ismerjünk meg egy másik megoldást: próbáljuk beállítani a tábla *IndexFieldNames* jellemzőjét az adott mező(-k)re, ha azonban ez az index nem létezik, akkor egy *EDatabaseError* típusú kivétel keletkezik.

```
With DM.tblCust Do
  begin
    Try
      IndexFieldNames:= 'City'; {ez egy nem létező index,
            emiatt átkerül a vezérlés a kivételkezelő blokkba}
      If Not FindKey(['Vancouver']) Then
        ShowMessage('Nincs találat!');
    Except
      On EDatabaseError Do
```

```
            ShowMessage('A tábla nem rendelkezik ''City'''+
                        'szerinti indexszel!');
      End;
   end;
```

- **FindNearest** (csak *TTable* komponenseknél használható): hozzávetőleges keresést valósít meg az aktuális index oszlopaiban. Mindig van találat, ez az első olyan rekord, melyben az indexbeli első mező értéke nagyobb vagy egyenlő a keresett értékkel.

  Például:

```
{Keressünk rá az első 'B' betűvel kezdődő nevű cégre. Ha nincs egy
'B' betűs sem, akkor egy 'C' betűs is jó...}
 With DM.tblCust Do
  begin
     Try
        IndexFieldNames:= 'Company';
        FindNearest(['B']);
     Except
        On EDatabaseError Do
           ShowMessage('A tábla nem rendelkezik ''Company'''+
                       ' szerinti indexxel!');
     End;
  end;
```

Futtassa le a 8_KERESES\PSEARCH.DPR alkalmazást. Benne élőben is kipróbálhatja a bemutatott keresési módszereket.

8.4.6 Egy adathalmaz szűrése

Adatbázisos alkalmazásainkban gyakran találkozunk azzal a feladattal, hogy egy adathalmaz tartalmát egy bizonyos feltétel szerint meg kellene szűrni (ha például csak az e havi megrendeléseket szeretnénk megjeleníteni). Erre a célra használhatjuk az adathalmazok **Filter**[1], **Filtered...** jellemzőit, de szűrhetünk egy SQL lekérdezéssel is. Lekérdezésekkel a 11. fejezetben foglalkozunk. Most tekintsük át a beépített szűrési lehetőségeket:

Egy adathalmaz (*TTable, TQuery, TStoredProc*) szűrési feltételét vagy a **Filter** jellemzőjébe, vagy az **OnFilterRecord** eseményjellemzőjébe írhatjuk be. A *Filter*-ben specifikált feltételt a mezőnevek segítségével fogalmazzuk meg, mint egy SQL utasításban

[1] Ez a szűrési lehetőség csak a 32 bites Delphi verziókban van jelen. A 16 bites Delphiben a *SetRange*, *CancelRange* metódusok állnak rendelkezésünkre, azonban ezek csak az aktuális index mezőire tartalmazhatnak értékhatárokat (akárcsak a *FindKey* és *FindNearest* metódusoknál is), és természetesen (az index miatt) csak *TTable* komponenseknél érhetők el. (Bővebben lásd a 8_SZURES\PFILTER.DPR alkalmazásban.)

(Például: *Country = 'Hungary'*). Az *OnFilterRecord* egy eseményjellemző, benne tehát akár elágazásokat, iterációkat is „bevethetünk" a szűrési feltétel megvizsgálására (az **Accept** változó paramétert kell beállítanunk Igaz vagy Hamis értékre, jelezvén, hogy az aktuális rekordot átengedjük-e, vagy pedig „fennakadt" a szűrőnkben).
Mindkét szűrési módszer (*Filter* vagy *OnFilterRecord*) esetében a szűrés csak addig érvényes, amíg a **Filtered** jellemző Igaz értékkel rendelkezik. Ekkor minden egyes rekord esetén a rendszer megvizsgálja a feltételt, és csak akkor engedi ezt át, ha a feltétel Igaznak bizonyul.

Figyelem! Ha egy adathalmaznál mind a *Filter*, mind az *OnFilterRecord* szűrőfeltételt tartalmaz, és ugyanakkor a *Filtered* jellemző Igaz értékű, akkor dupla szűrésünk van. Ekkor csak azokat a rekordokat láthatjuk, amelyek mindkét feltételnek eleget tesznek.

A karakterlánc típusú mezők esetén beállíthatunk bizonyos szűrési opciókat is a **FilterOptions** jellemző segítségével. Ez egy halmaz típusú jellemző, melynek maximálisan két eleme lehet: *[foCaseInSensitive, foNoPartialCompare]*.

Ha a *foCaseInSensitive* érték eleme a *FilterOptions* halmaznak, akkor a kis és nagy betűk nincsenek megkülönböztetve.

Ha a *FilterOptions* halmaznak nem eleme a *foNoPartialCompare*, akkor a feltételben használt '*' karakter Joker karakterként lesz értelmezve (akárcsak az SQL szabványban a '%' karakter). Ellenkező esetben a '*' ténylegesen csillag karakterként viselkedik (az SQL-ben ezt az *Escape* karakterekkel érhetjük el, lásd a következő példában).

Tekintsünk át néhány példát:

```
{Az USA-beli vevők}
Filter := 'Country = ''US''';

{Az 1998-as évben is aktív USA-beli vevők (akik 1998 január 1. után is
rendeltek)}
Filter := '(Country = ''US'') And (LastInvoicedate >= 1998.01.01)';

{Az USA-beli vagy magyarországi vevők}
Filter := '(Country = ''US'') Or (Country = ''Hungary'')';
```

Tekintsünk át egy példát a Joker karakterek használatára is. Végezzünk egy kis agytornát! Készítsük el a megfelelő szűrést SQL utasítással is:

A 'B' betűs országokból származó vevők (Bahamas, Belize, Bermuda...); a '*' és '%' Joker karakterként viselkedik:

Filter	SQL

`With DM.tblCust Do`
`  begin`
`  FilterOptions:= FilterOptions -`
`  [foNoPartialCompare];`
`  Filter := 'Country = ''B*''';`
`  Filtered:= True;`
`  end;`
	`SELECT *`
	`FROM Customer`
	`WHERE country LIKE 'B%'`

A '*' és '%' nem-Joker karakterként viselkedik, ha a következőket tesszük:

Filter	SQL

`With DM.tblCust Do`
`  begin`
`  FilterOptions:= FilterOptions +`
`  [foNoPartialCompare];`
`  Filter := 'Country = ''B*''';`
`  Filtered:= True;`
 	
`  end;`
	`SELECT *`
	`FROM Customer`
	`WHERE country LIKE 'B#%'`
	`ESCAPE '#'`
	
	{Az Escape utáni karakter (#) jelzi, hogy a feltételben utána következő karaktert (%) nem-Jokerként kell értelmezni}

A szűrési feltétel megadása azt jelenti, hogy csak a feltételnek megfelelő rekordok jelennek meg az eredményhalmazban. **Ennek az adatfelvitelre nincs semmi hatása.** Továbbra is felvihetünk egy új rekordot – természetesen, ha ezt az adatmodell megengedi –, ez beíródik az adathalmazba, legfeljebb, ha nem felel meg a szűrőfeltételnek, akkor postázása után rögtön eltűnik, mivel fennakad a szűrőn. Ezt úgy fogalmazhatjuk meg egyszerűen, hogy az itt megadott szűrés **egyirányú**: csak olvasásnál érvényes, írásnál nem. Ugyanez a helyzet egy *SELECT* lekérdezés esetén is[1]. **Kétirányú szűréseket** csak az adatbázisszerverek által támogatott nézetekkel (*VIEW*) hozhatunk létre, ott is csak egy külön opció (a *CHECK OPTION*) bekapcsolásával.

Ha nem áll módunkban nézeteket létrehozni, és ugyanakkor a programlogika megkívánja a kétirányú szűrést, akkor programkódból védhetjük le a szűrőfeltételnek meg nem felelő rekordok felvitelét. Ezt megtehetjük például az adathalmaz *BeforePost* eseményébe beépített ellenőrzéssel (lásd következő pont).

---

[1] A Delphi 32 bites verzióiban a *TQuery* komponens rendelkezik egy *Constrained* jellemzővel, mellyel befolyásolhatjuk a szűrés kétirányúságát. Ez azonban csak Paradox és dBase táblák estén működik.

Egyirányú szűrés

```
{Látni csak a magyar vevőket, fel-
vinni azonban nem magyarokat is
lehet}

{táblaszűréssel}
With DM.tblCust Do
   begin
      Filter:= 'Country = ''Hungary''';
      Filtered:= True;
   end;

{lekérdezéssel}
SELECT *
FROM Customer
WHERE Country = 'Hungary'

{nézettel}
CREATE VIEW Magyarok AS
SELECT * FROM Customer
WHERE Country = 'Hungary'
```

Kétirányú szűrés

```
{Látni is és felvinni is
csak a magyar vevőket
lehet}

{nézettel}
CREATE VIEW Magyarok AS
SELECT * FROM Customer
WHERE Country = 'Hungary'
WITH CHECK OPTION
```

Tekintsünk át néhány példát az ***OnFilterRecord*** eseményjellemző használatára is:

Az *OnFilterRecord* eseményre épített metódusban nem szabad semmi olyan műveletet végezni, ami a metódus újbóli hívásához vezethetne. Például nem szabad a *FilterOptions* jellemzőt átállítani, nem szabad az adathalmaz más rekordjára átlépni, nem szabad az aktuális rekordot szerkeszteni, postázni...

```
{csak az USA-beli vevőket engedjük át. Ezt a szűrést a Filter jellem-
zővel is megvalósíthatjuk.}
procedure TDM. tblCustFilterRecord1(DataSet: TDataSet;
                                    var Accept: Boolean);
begin
  Accept := Dataset.FieldbyName('Country').Value = 'US';
end;

{csak az eOrszag szerkesztődoboz által megadott országbeli vevőket
engedjük át. Ez a szűrés már mindenképpen némi programozást igényel.}

procedure TDM. tblCustFilterRecord2(DataSet: TDataSet;
                                    var Accept: Boolean);
begin
  Accept := Dataset.FieldbyName('Country').Value = eOrszag.Text;
end;
```

```
{csak azokat a vevőket akarjuk látni, akiknek a Fax számuk megegyezik
a telefonszámukkal}
procedure TDM. tblCustFilterRecord1(DataSet: TDataSet;
                                    var Accept: Boolean);
begin
   Accept := Dataset.FieldbyName('Fax').Value =
             Dataset.FieldbyName('Phone').Value;
end;
```

Az adathalmaz azonos rekordbeli mezőértékeinek összehasonlítását csak az *OnFilterRecord* metódusba építhetjük be, ezt a *Filter* jellemző nem fogadná el (a *Filter := 'Fax = Phone'* beállítás az „*Operation Not Applicable*" hibához vezetne).

```
{Azokat a vevőket szeretnénk látni csak, akik az utolsó hónapban is
vásároltak tőlünk. Ennek érdekében kikeressük, hogy melyik az utolsó
vásárlás hónapja (MaxHo) (nem az aktuális hónapot vesszük, hiszen
lehet, hogy végig leltár volt az utolsó hónapban), majd a MaxHo havi
vásárlókat engedjük csak át.}
procedure TDM.tblCustFilterRecord2(DataSet: TDataSet;
                                   var Accept: Boolean);
var  MaxDatum:TDateTime;
     Ho, MaxHo, Ev, MaxEv, Nap:Word;
begin
   MaxDatum:=0;
   With tblCust2 Do
      begin
         First;
         While Not EOF Do
            begin
               if Fieldbyname('LastInvoiceDate').Value > MaxDatum Then
                  MaxDatum:=Fieldbyname('LastInvoiceDate').Value;
               Next;
            end;
      end;
   DecodeDate(tblCust.Fieldbyname('LastInvoiceDate').AsDateTime,
              Ev, Ho, Nap);
   DecodeDate(MaxDatum, MaxEv, MaxHo, Nap);
   Accept:= (Ev = MaxEv) And (Ho = MaxHo);
end;
```

Ez az utolsó példa annyira csúnya és lassú, hogy tekintsük inkább ***ellenpéldának***. Ezt a feladatot tipikusan egy SQL lekérdezéssel kellene megoldani. A rossz példából is lehet viszont tanulni. Ebben például azt vesszük észre, hogy egy másik *TTable* komponens rekordjain szaladunk végig a *MaxDatum* kiválasztására. Hát persze, hiszen az éppen szűrés alatt álló *tblCust* aktuális rekordját nem szabad megváltoztatnunk. Ezért kénytelenek voltunk egy második, ugyancsak a CUSTOMER.DB fizikai állományra irányuló tábla komponenst is elhelyezi adatmodulunkon, a *tblCust2*-t.

Futtassa le a 8_SZURES\PFILTER.DPR alkalmazást! Próbálja ki segítségével élőben is a bemutatott szűrési lehetőségeket!

8.4.7 Adathalmazok eseményei

Az adathalmazok eseményei *BeforeValami* (Valami előtt) és *AfterValami* (Valami után) formájúak. Például *BeforeOpen – AfterOpen*, *BeforePost – AfterPost*... Vegyük ezeket sorban:

- **BeforeOpen, AfterOpen**: adathalmaz megnyitása előtt és után
- **BeforePost, AfterPost**: egy rekord lementése (postázása) előtt és után. Használhatjuk mentésre való rákérdezésre, frissítésekre, rekordszintű ellenőrzésekre. Például, ha tanfolyamok esetén tároljuk ezek kezdeti- és végdátumait; minden adatmentés előtt rekordszinten le kell ellenőriznünk, hogy a megadott kezdeti dátum kisebb vagy egyenlő a végdátummal. Ha a feltétel nem teljesül, akkor az *Abort* metódussal megszakítjuk a mentési folyamatot.

```
procedure TDM.tblTanfolyamBeforePost(DataSet: TDataSet);
begin
  With DataSet Do
     begin
        If FieldByName('KezdDatum').Value >
           FieldByName('VegDatum').Value Then
           begin
              ShowMessage('Nem jó dátummegadás!');
              Abort;
           end;
     end;
end;
```

- **BeforeDelete, AfterDelete**: az aktuális rekord törlése előtt és után következnek be. Ekkor használhatjuk törlés helyességének megvizsgálására és törlés jóváhagyására (lásd a 8.4.4. pontban).
- **OnNewRecord**: új rekord felvitelekor következik be. Használhatjuk kezdőértékek beállítására. (De ha egy mód van rá, akkor a kezdőértékeket az adatmodellbe építsük be.) Példánkban egy új megrendelés felvételénél az eladás dátumát beállítjuk a mai dátumra.

```
procedure TDM.tblMegrendNewRecord(DataSet: TDataSet);
begin
  Dataset.FieldByName('SaleDate').Value:= Date;
end;
```

- **OnCalcFields**: ha adathalmazunkban vannak számított mezők, akkor minden egyes megjelenítendő rekord esetén meghívódik az *OnCalcFields* metódus; az ide beírt utasítások alapján fogja a program kiszámolni a számított mező aktuális rekordbeli értékét. (Bővebben lásd a 8.5. pontban.)

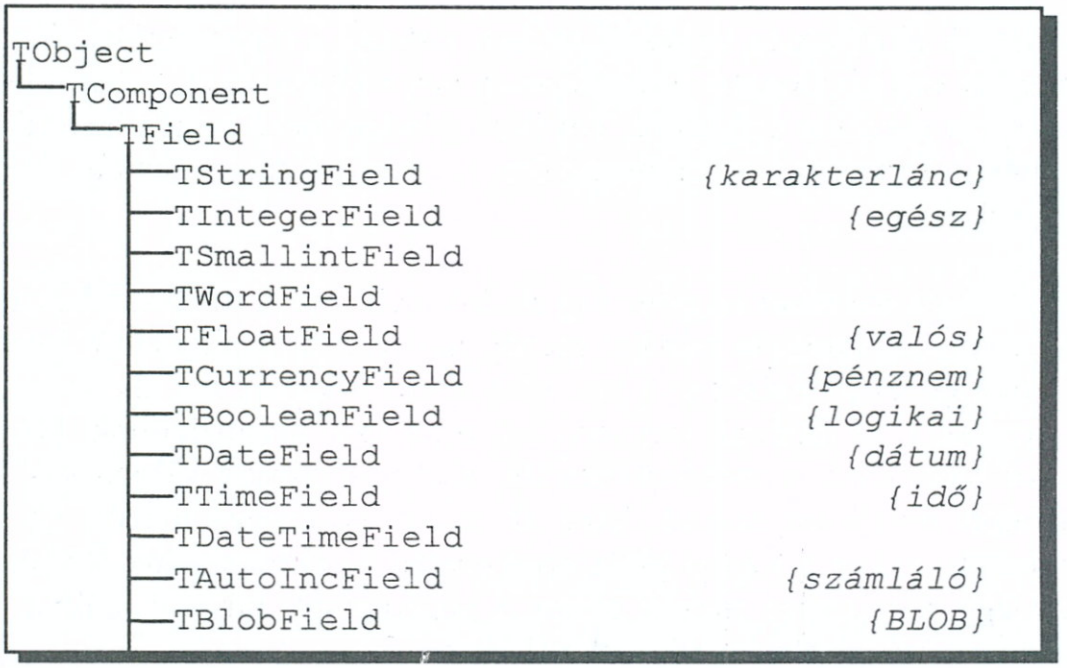
- **OnEditError, OnPostError, OnDeleteError**: szerkesztéskor, postázáskor vagy törléskor észlelt hiba esetén következnek be. Paraméterként megkapják a hiba által generálódott kivételobjektumot (*E:EDatabaseError*) és azt az adathalmazt, amelyben a hiba megtörtént (*DataSet:TDataSet*). A hiba lekezelése a mi kezünkben van: az *Action:TDataAction* változó paramétert a következő értékek valamelyikére kell beállítanunk:
  - *daFail*-re, ha azt szeretnénk, hogy az alapértelmezett (angol) hibaüzenet jelenjen meg.
  - *daAbort*-ra, ha „csendben akarjuk elintézni az ügyet". Ilyenkor nem fog megjelenni üzenet a képernyőn, hacsak mi explicit módon meg nem jelenítünk egyet (például az egyszerű felhasználók számára is érthető magyar üzenetet).
  - *daRetry*-ra, ha tudjuk, mi okozta a hibát, és azt el is tudjuk hárítani. A hibaelhárítás után újra próbálkozunk az *Action:=daRetry* beállítás következtében.

Például gondoljunk vissza a 8.4.4. pontban tárgyalt esetre: egy megrendelést mindaddig nem törölhetünk ki, amíg vannak tételei (lásd a 8.7. ábrán). A *tblOrdersOnDeleteError* metódus akkor fog meghívódni, ha egy megrendelés törlése meghiúsult. Ekkor kínáljuk fel a rendelés tételeinek törlését. Ha a felhasználó igennel válaszol, akkor letöröljük a tételeket, majd újrapróbálkozunk a megrendelés kitörölésével.

```
procedure TDM.tblOrdersOnDeleteError(DataSet: TDataSet;
                E: EDatabaseError; var Action: TDataAction);
begin
  if Messagedlg('A megrendelést nem lehet kitörölni, mert'+
             ' vannak tételei. Töröljük ki a tételeket is?',
            mtWarning, [mbYes, mbNo, mbCancel],0) = mrYes Then
     begin
        With DM.tblItems Do
           begin
              {kiválogatjuk csak az adott megrendelésnek a
              tételeit, majd azokat letöröljük mind egy szálig}
              Filter:= 'Orderno = ' +
                   DM.tblOrders.Fieldbyname('Orderno').AsString;
              Filtered:=True;
              First;
              While Not EOF Do
                 Delete;
              Filtered:=False;
           end;
        Action:=daRetry;
     end
  else
     Action:=daAbort;
end;
```

8.5 Az adathalmazok mezői. A TField osztály

Egy adathalmaz megnyitásakor a rendszer automatikusan minden mezője számára létrehoz egy-egy *TField* objektumot. A BDE a mező deklarált típusától függően kiválasztja a megfelelő mezőosztályt.



8.8. ábra. Mezőosztályok

Például: vegyünk fel egy *Alkalmazottak* táblát, melyben tároljuk minden dolgozónk azonosítóját (számláló[1]), nevét (*String[50]*), születési dátumát (dátum), nemét (logikai) és fényképét (BLOB[2]). A tábla mezőinek Delphiben a következő objektumok felelnek meg: *AlkalmazottAz:TAutoIncField*, *Nev:TStringField*, *SzulDatum:TDateField*, *Nem:TBooleanField*, *Fenykep:TBlobField*.

A mezőobjektumokat vagy a rendszer hozza létre automatikusan (ezek lesznek a **dinamikus mezők, *dynamic fields***), vagy pedig mi (így keletkeznek a **perzisztens mezők, *persistent fields***). Ha az automatikus létrehozás mellett döntünk (ez az alapértelmezés), akkor a rendszer az adathalmaz megnyitásakor minden egyes mező számára automatikusan generál egy mező-objektumot. Ennek a megoldásnak az előnye az, hogy így a mezők mindig követik a fizikai tábla szerkezetét. Ha egy mezőt kitörlünk, egy másikat meg felveszünk a táblába, akkor a Delphi programunk ezt követni fogja. Az automatikus mezőgenerálás hátrányaként értelmezhető viszont az, hogy, ha például egy adott pillanatban egy

---

[1] A *számláló (autoincrement)* típusú mezők értékeit a program generálja, figyelve a szolgáltatott értékek egyediségére.

[2] A *BLOB* a *Binary Large Object* rövidítése; a *BLOB* típusú mezőkben bármilyen bináris adatot tárolhatunk (grafikát, hangot...).

adathalmazból csak két mezőre lenne szükségünk a tizennégyből, akkor is alkalmazásunk minden rekordnál beolvasná mind a tizennégy mező értékét. Ezáltal programunk lelassulna, és a memóriát is fölöslegesen fogyasztaná. Egy másik – kisebb – hátránya ennek a megoldásnak az, hogy – mint később látni fogjuk – így körülményesebben hivatkozhatunk a mezők értékeire.

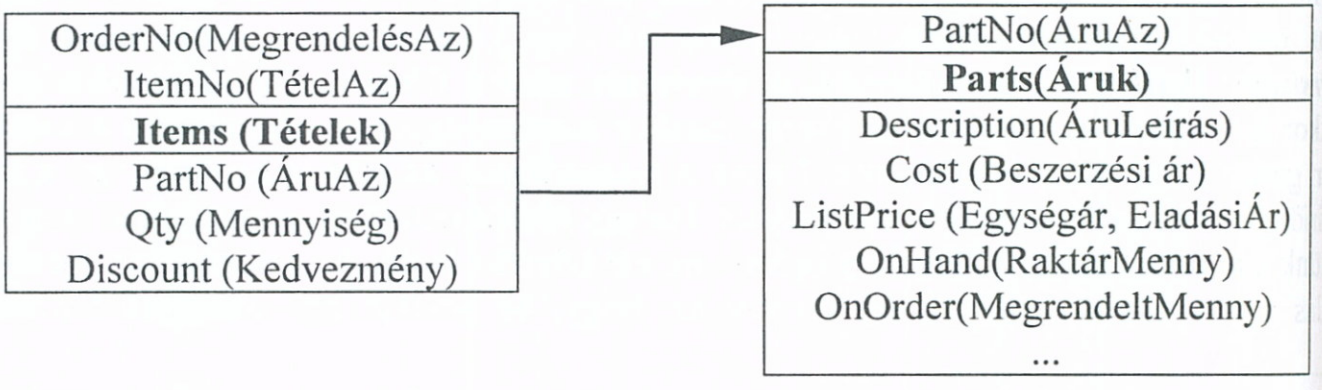
A perzisztens mezők létrehozása tervezési időben történik a mezőszerkesztő segítségével. Ekkor lehetőségünk van kiválogatni a ténylegesen használt mezőket, valamit lehetőségünk van új, származtatott mezőket definiálni. Továbbá minden mező esetén beállíthatunk bizonyos megjelenítési, szerkesztési tulajdonságokat is, mint például: igazítás, szélesség, formátum, és még sok más.

Egy dologra azonban nagyon figyelnünk kell! Ha a mezőszerkesztővel felvettünk egy adathalmazba néhány mezőt, akkor ettől a pillanattól kezdve a rendszer már egyetlen mezőt sem hoz létre automatikusan. Az alkalmazás éles futásakor csak a mi általunk megtervezett mezőkkel dolgozhatunk. Ha tehát tervezéskor használjuk a mezőszerkesztőt, akkor gondosan válogassuk ki, és származtatott mezőként hozzuk létre az összes szükséges mezőt. És ha már ezt megtettük, akkor a fizikai tábla szerkezetén se módosítsunk többé: ha kitörölnénk egy perzisztens mezőt, akkor a programunk hibával leállna, mivel nem létező mezőre tartalmazna hivatkozásokat.

8.5.1 A mezőszerkesztő használata

A mezőszerkesztő külalakja és használata eltérő a Delphi 16 és 32 bites verzióiban. A következő gyakorlatban a 32 bites verzió képeit láthatjuk; bízom abban, hogy a 16 bites Delphivel rendelkező Olvasók adaptálni tudják az itt olvasottakat saját környezetükre. Ebben segítségükre van a feladat megoldása is (🖫 8_MEZOK\PFIELDS.DPR).

A mezőszerkesztő tanulmányozására egy kis alkalmazást fogunk elkészíteni. Ismerkedjünk meg előbb a benne használt adatokkal:



8.9. ábra. Az adatmodell

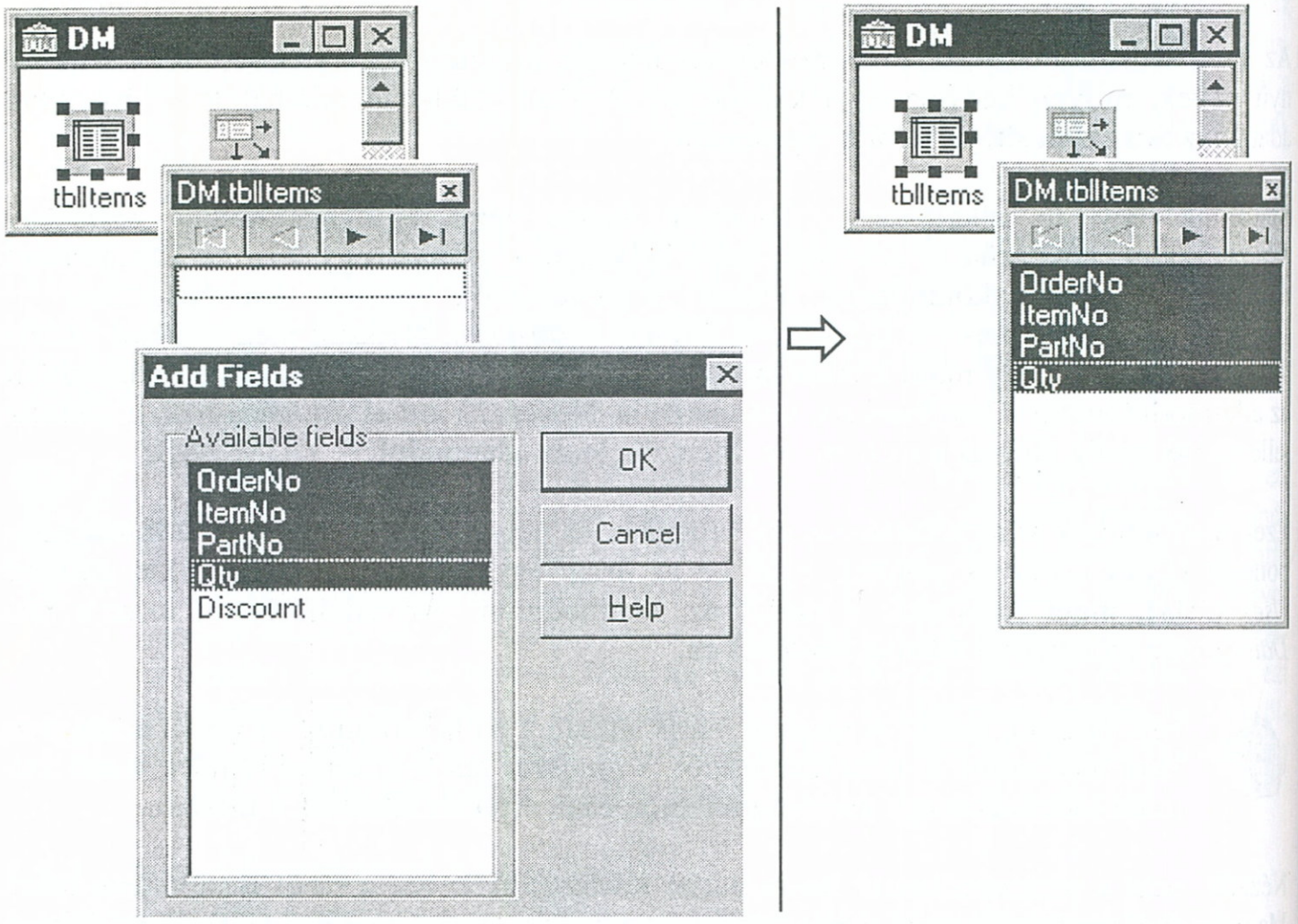
Az *Items* táblában tároljuk a megrendelések tételeit: melyik megrendelésen, miből, mennyit vettek, milyen kedvezményt kaptak stb. A *Parts* táblában találhatók az árucikkek adatai: az áru azonosítója, leírása, egységára...

Kezdjünk egy új alkalmazást. Hozzunk benne létre egy adatmodult *(File/New DataModule)*, nevezzük el *DM*-nek, majd helyezzünk el rajta egy *TTable* komponenst. Állítsuk be a táblakomponens adatait a következők szerint: *DatabaseName := DBDEMOS, TableName := ITEMS.DB, Name := tblItems*. Nyissuk meg a táblát (*Active := True*). Ahhoz, hogy majd a tábla mezőértékeit meg is jeleníthessük, el kell helyeznünk az adatmodulon egy *DataSource* komponenst is. Nevezzük ezt el *dsrItems*-nek, a *DataSet* jellemzőjét pedig állítsuk a *tblItems*-re. Mentsük le az adatmodult ⇨ UDM.PAS.

Ezek után lépjünk át az alkalmazásunk űrlapjára, és helyezzünk el rajta egy *DBGrid* komponenst (*Data Controls* paletta). Írjuk be az űrlap egységének *Implementation* részébe: *Uses UDM*, majd irányítsuk a rácsot az adatmodulon levő adatforrásra, azaz a rács *DataSource* jellemzőjét állítsuk *dsrItems*-ra.

Mit veszünk észre? A rácsban máris látható a tábla tartalma. És mivel eddig még nem használtuk a mezőszerkesztőt, a rendszer automatikusan minden mező számára létrehozott egy mezőobjektumot; ezek értékeit láthatjuk a rácsban.

Nézzük most a mezőszerkesztő használatát! Kattintsunk kettőt a tábla komponens felett. Megjelenik a mezőszerkesztő párbeszédablak, mely egyelőre üres (lásd 8.10. ábra). Hívjuk meg az egér jobb gombjával a gyorsmenüt: az *Add fields...* paranccsal a fizikai táblában létező mezőket olvashatjuk be, míg a *New field...* paranccsal új, származtatott mezőket hozhatunk létre. Hívjuk meg az *Add fields...* parancsot, és jelöljük ki az összes mezőt a *Discount* kivételével (tételezzük fel, hogy erre most nincs szükségünk). A mezőszerkesztőben csak a kijelölt mezők fognak megjelenni. A nyilak segítségével navigálhatunk a táblában.

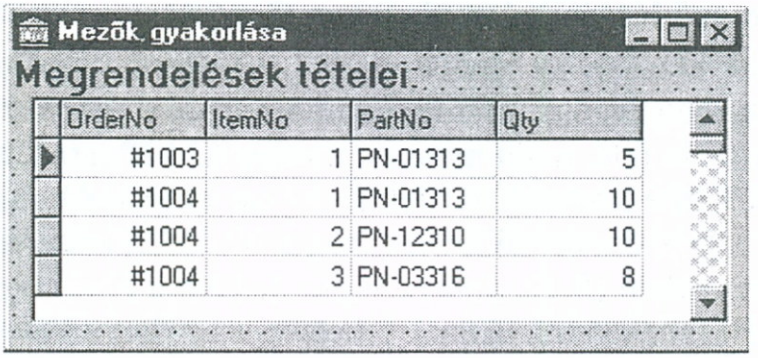


8.10. ábra. A mezőszerkesztő használata

Ha most megfigyeljük a rács tartalmát, akkor tapasztalni fogjuk, hogy a Discount oszlop eltűnt. A rendszer már nem hozza ezt létre, mivel mi „kigyomláltuk" a táblából (lásd 8.11. ábra).

Még egy érdekességre fel szeretném hívni a kedves Olvasó figyelmét: ha a mezőszerkesztőben kijelölünk egy mezőt, akkor az objektum-felügyelő címsorában a mezőobjektum nevét és típusát látjuk (például *tblItemsOrderNo:TFloatField*), alatta pedig a mező tervezési időben is állítható tulajdonságait (ezeket később, a 8.5.3. pontban tárgyaljuk). Nézzük meg az adatmodul osztálydefinícióját; azt vesszük benne észre, hogy a perzisztens mezők az adatmodul osztályának mezőiként lettek definiálva.

```
type TDM = class(TDataModule)
        tblItems: TTable;
        dsrItems: TDataSource;
        tblItemsOrderNo: TFloatField;
        tblItemsPartNo: TFloatField;
        ...
    end;
...
```



8.11. ábra. A *Discount* oszlop eltűnt!

8.5.2 Származtatott mezők létrehozása

Adatmodelljeikben mindig kerüljük a redundanciát[1]. Ha például egy áru leírását az *Áruk* táblában tároljuk, akkor fölösleges azt a *Tételek* táblában is megtenni (elég a tételeknél az áru azonosítóját tárolni, ennek alapján bármikor kikereshetjük az áru egyéb adatait). Igen ám, de a tételek megjelenítésekor jó lenne látni az áru leírását is (a 8.11. ábrán a PN-01313 nem túl sokat mond). Ennek érdekében a *Tételek* táblakomponensben létre fogunk hozni egy származtatott mezőt. Ugyancsak származtatott mezőre van szükség akkor is, ha az egyes árucikkekből rendelt mennyiség, a kapott kedvezmény (%) és az áru egységára alapján meg szeretnénk jeleníteni a tétel összértékét is:

egy tétel összértéke = mennyiség * egységár * (1 - kedvezmény/100).

Természetesen ezeket a mezőket nem lehet majd szerkeszteni, módosítani, ezek csak megjelenítési célokat szolgálnak.

Delphiben egy származtatott mező kétféle lehet:

- **„Kikeresett" mező (*Lookup field*)**: azt a mezőt nevezzük így, melynek értékét egy másik táblából nézzük ki. Ilyen lesz az *ÁruLeírása*, mivel ezt az *Áruk* táblából fogjuk kikeresni a *Tételek* táblában levő *ÁruAz* alapján.
- **Számított mező (*Calculated field*)**: értéke nem található meg egy másik táblában sem, valamilyen képlet alapján kell kiszámolnunk. Számított mezőként fogjuk a tétel összértékét kiszámolni.

A 32 bites Delphi környezetekben a „kikeresett" mezőknél elég megadnunk egy párbeszédablakban a kikeresés módját (melyik táblából, milyen mező alapján). A számított mezőknél a tábla *OnCalcFields* eseményjellemzőjébe kód formájában be kell építenünk a számolási képletet.
A 16 bites Delphiben minden származtatott mezőnél kódolnunk kell, ilyen értelemben tehát nincs különbség a „kikeresett" és a számított mezők között.

---

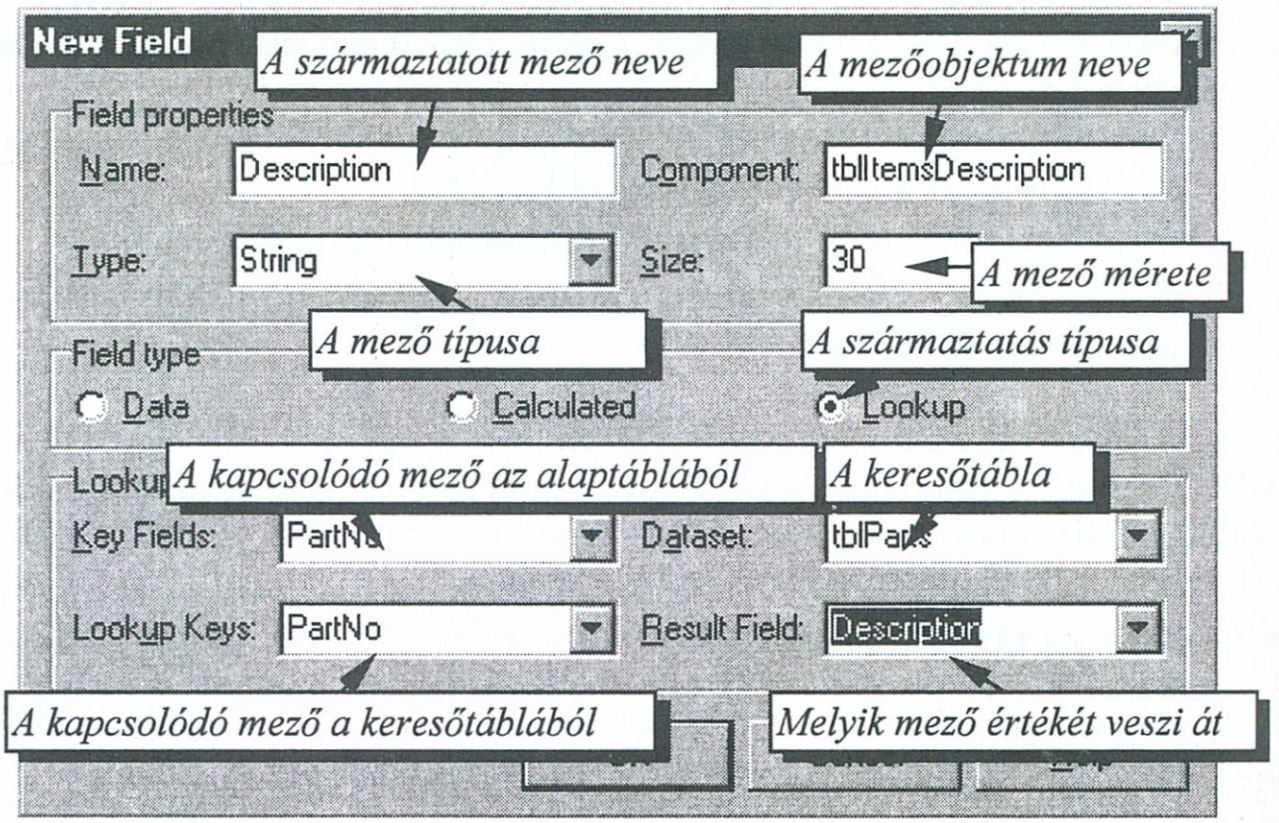
[1] Redundancia = feleslegesség. Ugyanazon adatok ismételt tárolása, vagy létező adatok alapján is kiszámítható értékek tárolása.

8.5.2.1 „Kikeresett" mező létrehozása (*Lookup field*)

Jelenítsük meg a *Tételek* rácsban az *ÁruLeírását* (*Description*). Mivel ennek értéke az *Áruk* táblából származik, helyezzünk el adatmodulunkra egy újabb *TTable* komponenst, irányítsuk a PARTS.DB tábla felé, majd nyissuk is meg. A **tblParts** tehát a **keresőtábla**. Kattintsunk duplán a ***tblItems*** komponens felett (ez **az alaptábla**, hiszen ebben hozzuk létre az új mezőt), majd a gyorsmenüből hívjuk meg a *New field...* parancsot.
A megjelenő párbeszédablakban (8.12. ábra) a legfontosabb a *Field type* szekció, melyben a származtatás típusát kell megadnunk. Ennek lehetséges értékei a következők:

- *Data:* ezt akkor használjuk, amikor az új létrehozandó mező megtalálható a fizikai táblában, de például más típusúnak szeretnénk felvenni, mint amilyennek a rendszer automatikusan létrehozná. Nagyon ritkán használjuk, így itt ezt az esetet nem tárgyaljuk.
- *Calculated:* számított mező
- *Lookup:* valamely keresőtáblából kikeresett mezőérték

Az *ÁruLeírás* (*Description*) mező esetében a *Lookup* választógombot kell bejelölnünk. Töltsük ki a többi mezőt is az ábrának megfelelően.



8.12. ábra. Új mező létrehozásának párbeszédablaka

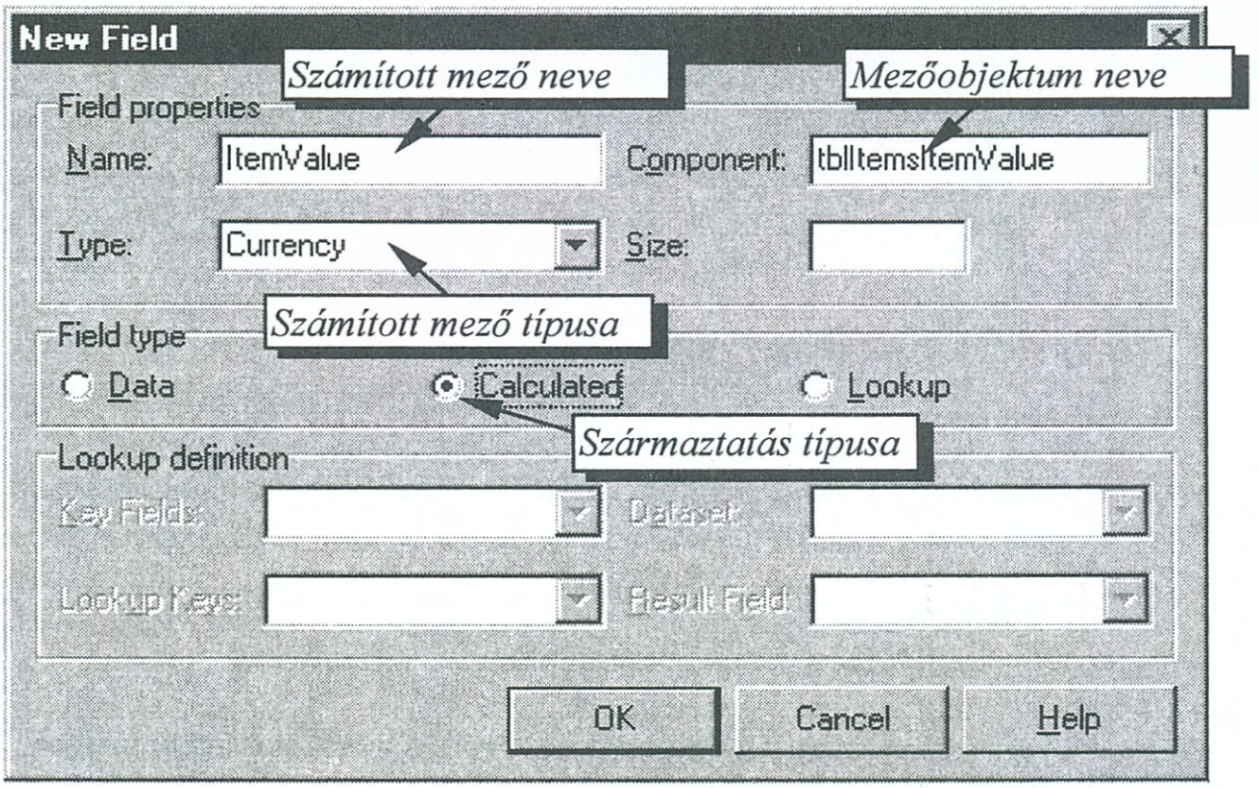
A párbeszédablak bezárása után azt tapasztaljuk, hogy az új mező megjelenik a tételek rácsban, értékei ki is töltődnek (az adatok élnek már tervezési időben is).

8.5.2.2 Számított mező létrehozása (*Calculated field*)

Számoljuk ki minden egyes megrendelés tételeinek összértékét a következő képlet alapján: *TételÖsszérték:=Mennyiség*Egységár*(1-Kedvezmeny/100)*, vagy az angol mezőnevekkel: *ItemValue:=Qty*ListPrice*(1-Discount/100)*.

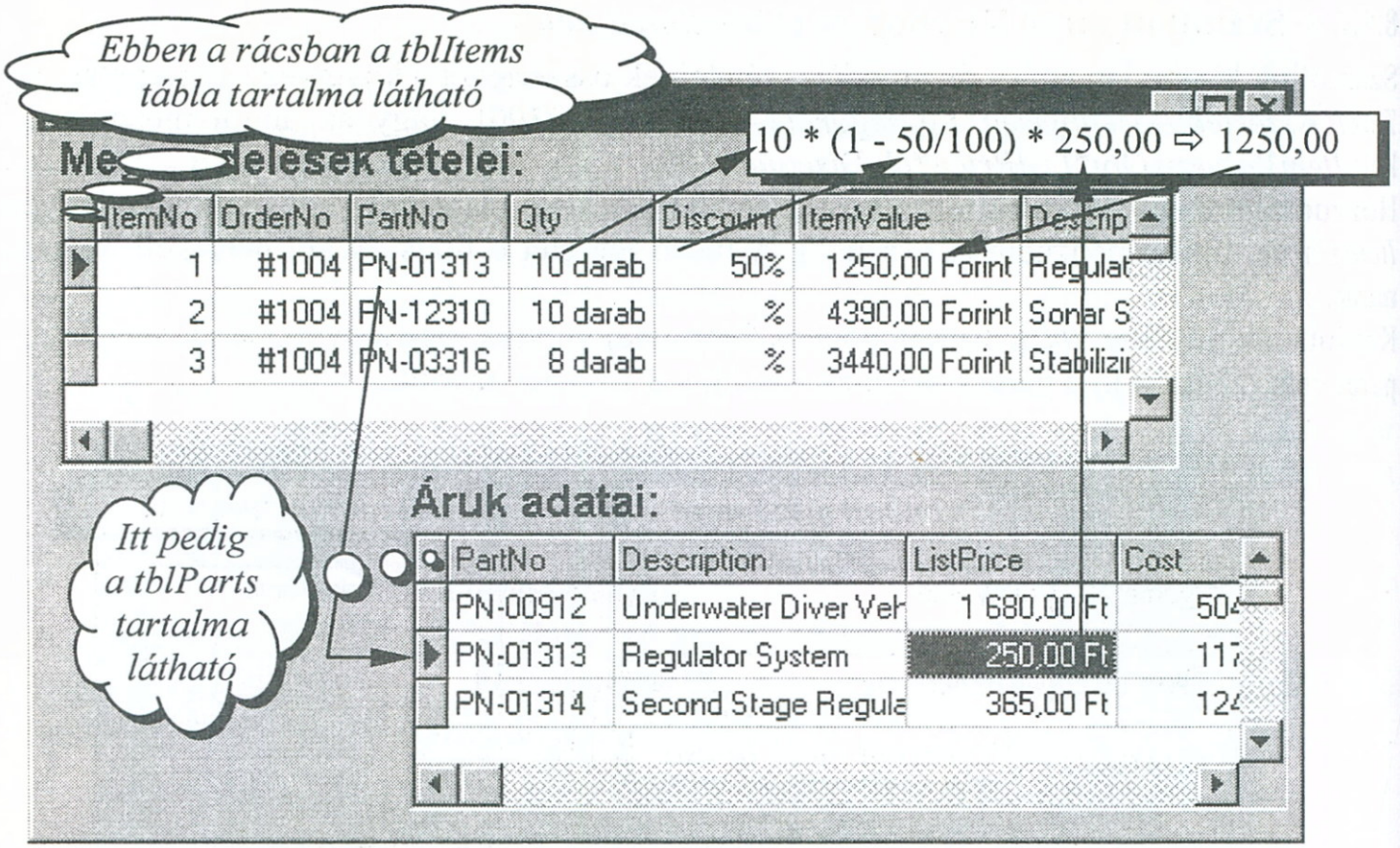
Hozzunk létre egy új, számított mezőt a *Tételek (tblItems)* táblában, melynek neve legyen *ItemValue*, típusa pénznem (*Currency*). A mező méretét e típus esetén nem kell megadnunk.

Kattintsunk duplán a *tblItems* komponens felett, majd hívjuk meg a gyorsmenü *New field...* parancsát. Töltsük ki a párbeszédablak mezőit a 8.13. ábra szerint.



8.13. ábra. Számított mező létrehozása

Az új mező kiszámolásának képletét a táblakomponens (*tblItems*) *OnCalcFields* eseményjellemzőjébe fogjuk beírni. Figyelem, a mennyiség és kedvezmény információ helyben megtalálható (a *tblItems*-ben), az egységárat viszont ki kell keresnünk a *tblParts* táblából. Ezt mutatja a 8.14. ábra is.



8.14. ábra. Az árucikk egységárát (*ListPrice*) a *tblParts* táblából nézzük ki

A kód a következő lesz:

```
procedure TDM.tblItemsCalcFields(DataSet: TDataSet);
var Egysegar:Variant;
begin
   With Dataset Do
      begin
         {Előbb kikeressük a tblParts táblából az áru egységárát
         (ListPrice), és ha megtaláltuk, akkor számolunk vele}
         Egysegar:= DM.tblParts.Lookup ('PartNo',
                              FieldByName('PartNo').Value, 'ListPrice');
         If Not VarIsNull(Egysegar) Then
            FieldByName('ItemValue').Value:= FieldByName('Qty').Value *
                  EgysegAr * (1- FieldByName('Discount').Value/100);
      end;
end;
```

Ez a számított mezőt tartalmazó adathalmaz (tblItems)

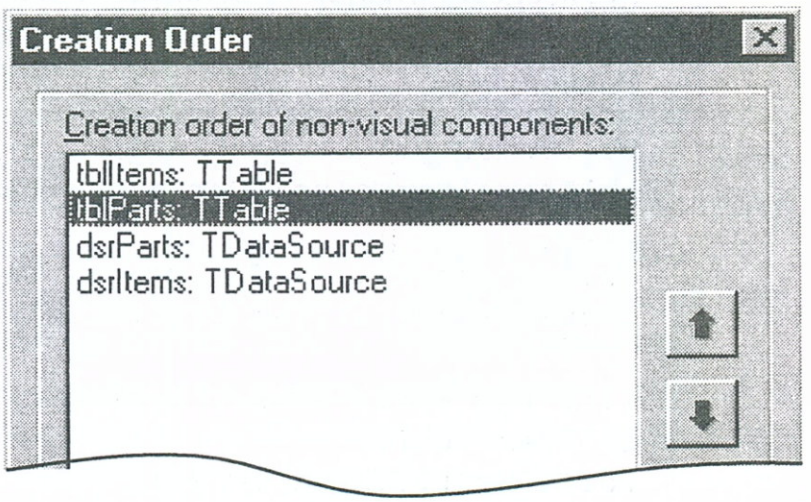
16 bites Delphiben az áru egységárának kikeresését csakis a *FindKey* metódussal tehetnénk meg, mivel nincs implementálva *Lookup* metódus. Ennek értelmében a kód a következő lenne (természetesen, mivel *FindKey* metódus a Delphi 32-ben is van, így a következő kódrészlet az újabb környezetekben is tökéletesen megfelel a célnak):

```
procedure TDM.tblItemsCalcFields(DataSet: TDataSet);
begin
  With Dataset Do
    begin
      If tblParts.FindKey([FieldByName('PartNo').Value]) Then
        Fieldbyname('ItemValue').Value:= FieldByName('Qty').Value *
                    tblParts.FieldByName('ListPrice').Value *
                    (1- FieldByName('Discount').Value/100);
    end;
end;
```

Mire kell figyelnünk a számított mezők létrehozásánál?

- Egy adathalmaz *OnCalcFields* eseményében csak **rövid kódrészlet**et szabad elhelyeznünk (mivel nagyon gyakran meghívódik); ráadásul abban **nem szabad semmi olyat tennünk, ami az esemény újbóli bekövetkezéséhez vezethetne**: nem szabad átlépnünk másik rekordra, nem szabad szerkesztenünk vagy postáznunk a rekordot... Ez az esemény minden egyes megjelenítendő rekordra külön-külön meghívódik, így benne csak az éppen aktuális rekordban levő számított mezők értékeit (és csak ezekét) kell kiszámítanunk.
- Ha az *OnCalcFields* eseményre épített metódusban hivatkozunk, keresgélünk egy másik adathalmazban, akkor figyelnünk kell arra, hogy a **kereső-adathalmaz ekkorra már nyitott legyen**. Példánkban a *tblItems* tábla megnyitásakor lefut az előbbiekben megírt *tblItemsCalcFields* metódus a megjelenítendő rekordokra. Ha ekkor a *tblParts* még nem lenne nyitva, akkor programunk még a legelején leállna a *„Cannot perform this operation on a closed dataset"* futási hibával. Ezt kétféleképpen is orvosolhatjuk:

1. **Első megoldás**: az adatmodul (vagy ha táblakomponenseink egy űrlapon vannak – Delphi 16 –, akkor az űrlap) gyorsmenüjéből hívjuk meg a ***Creation Order*** parancsot. Az így megjelenő párbeszédablakban beállíthatjuk a futáskor láthatatlan komponensek létrehozási sorrendjét. Cseréljük itt fel a *tblParts* és *tblItems* táblák sorrendjét.



8.15. ábra. A létrehozási sorrend beállítása

2. **Második megoldás**: zárjuk be a táblákat tervezési időben, és nyissuk meg őket kódból, az adatmodul (vagy Delphi 16-ban az űrlap) ***OnCreate*** metódusában. Így alkalmunk nyílik a helyes sorrendet specifikálni:

```
procedure TDM.DMCreate(Sender: TObject);
begin
  tblParts.Open;
  tblItems.Open;
end;

{Ha már mi nyitjuk meg a táblákat, akkor mi is zárjuk ezeket be}
procedure TDM.DMDestroy(Sender: TObject);
begin
  tblParts.Close;
  tblItems.Close;
end;
```

8.5.3 A mezőobjektumok jellemzői, eseményei

Minden mezőobjektum – akár tervezési, akár futási időben jön létre – rendelkezik az alábbi jellemzőkkel és eseményekkel. Ha egy mezőobjektum már tervezéskor megszületik, akkor tulajdonságai az objektum-felügyelőben megjelennek, ott szerkeszthetők.

- Fontosabb jellemzők:
  - **FieldName**: a mező táblabeli elnevezése (oszlop neve). Például *Qty*.
  - **Name**: a mezőobjektum neve. Ennek alapértelmezett értéke a táblanév és mezőnév konkatenálásával keletkezik. Legtöbbször ezt elfogadjuk. Például *tblItemsQty*
  - **IsNull: Boolean** nagyon hasznos jellemző, lekérdezhetjük vele, hogy a mező üres-e (nincs megadva az értéke), vagy pedig ki van töltve.
  - **Value: Variant** (csak a 32 bites Delphi verziókban)
    Segítségével írhatjuk és olvashatjuk a mező értékét. Az így kapott értéket a rendszer automatikusan konvertálja a környezet által elvárt típusra.
    Például:

```
tblItemsQty:TIntegerField;
tblItemsItemValue:TCurrencyField;
...
{előbb átalakítja String-gé, majd megjeleníti}
ShowMessage(tblItemsQty.Value);
...
{itt egy egész számot helyezünk el benne}
tblItems.Edit;
tblItemsQty.Value := 20;
tblItems.Post;
...
{itt pénznem típusú értéket számolunk ki}
tblItemsItemValue.Value:= tblItemsQty.Value *
tblPartsListPrice.Value *(1- tblItemsDiscount.Value/100);
```

- **AsString, AsInteger, AsBoolean, AsFloat, AsDateTime, AsCurency, AsVariant:** konverziós jellemzők, melyeket a mező értékének leolvasására és állítására használjuk. Ilyenkor – függetlenül az adat tényleges típusától – lekérdezhetjük és állíthatjuk értékét, mint egész számot (*AsInteger*), vagy mint karakterláncot (*AsString*) stb.
  Például legyen a *tblItemsQty:TIntegerField* mezőobjektum. Elemezzük a következő utasításokat (zárójelben a konverziókat tüntettük fel):

```
tblItemsQty.AsInteger:= 17;              {Integer 17⇒Integer 17}
tblItemsQty.AsString:= '17';            {String '17'⇒Integer 17}
blItemsQty.AsFloat:=17.3;                {Float 17.3⇒Integer 17}
blItemsQty.AsFloat:=17.8;                {Float 17.8⇒Integer 18}
ShowMessage(tblItemsQty.AsString);
tblItemsQty.AsInteger := tblItemsQty.AsInteger *10;

{a következő utasításban a konverzió sikertelen}
blItemsQty.AsString:='17.3';     {String 17.3⇒Integer 17.3 💣}
```

  Amikor az *AsFloat* jellemzőt használjuk, akkor a rendszer a *Float⇒Integer* konverzió hatására a kerekített értéket állítja be. Ha viszont az *AsString* jellemzővel próbáljuk értékét '17.3'-ra beállítani, akkor a *String⇒Integer* konverziót hajtja végre, a mi esetünkben pedig ez sikertelennek fog bizonyulni. Erre egy *EDatabaseError* kivétel is figyelmeztet.
  A 16 bites Delphiben a *TField* osztályban nincs implementálva *Value* jellemző, így **csak a konverziós jellemzőkkel hivatkozhatunk egy mező értékére.**

- **Alignment**: a mező értékének igazítása a megjelenítési komponensen (például szerkesztődobozon) belül
- **DisplayFormat**: a mező megjelenítési formátuma
  Például *DisplayFormat := 0 darab* esetén a mennyiség információt mindig követni fogja a *'darab'* szöveg. Egy nulla mennyiség esetén *'0 darab'* jelenne meg. Ha viszont a *'# darab'* formátumot adnánk meg, akkor nulla mennyiség esetén csupán *'darab'* jelenne meg. (A további variációkat lásd a súgóban.)
- **EditFormat**: a mezőérték szerkesztése közbeni formátumát határozhatjuk vele meg. A formátum megadása ugyanúgy történik, mint a *DisplayFormat*-nál.
- **DisplayLabel**: a mező címkéje. Ez fog megjelenni alapértelmezés szerint[1] a rácsok fejlécében, vagy akkor, ha a mezőt a mezőszerkesztőből rávonszoljuk egy űrlapra (ekkor a rendszer automatikusan létrehoz egy adatmegjelenítési komponenst (például *DBEdit*-et) és egy címkét, amiben a *DisplayLabel*-ben megadott szöveg látható).

---

[1] A mezők külalakját, fejlécét (címkéjét), szélességét a rács komponenseknél tovább állíthatjuk (lásd a 9. fejezetben). Így nyilván felülbírálhatjuk a mezőobjektum tulajdonságaiban beállított értékeket.

Állítsa be a *Qty* mező címkéjét *Mennyiség*-re, majd vonszolja ezt a mezőt a mezőszerkesztőből az alkalmazás űrlapjára. Mit tapasztal?

- **DisplayWidth**: a mező szélessége
- **EditMask**: egy párbeszédablak segítségével beállíthatjuk a mező bemeneti maszkját
  Például telefonszám esetén: *EditMask := !\(999\)000-0000;1;_*, ami például a következő telefonszámnak felelne meg: (415)555-1212. Az '1'-es azt jelenti, hogy a zárójelek és szóközök is tárolásra kerülnek, míg az '_' a helyettesítő karakter. Kitöltés előtt így néz ki: '(___)___-____'. Bővebben lásd a súgóban.
- **MinValue**, **MaxValue**: a mező megengedett értékeinek alsó és felső határa. Mezőszintű ellenőrzést valósíthatunk vele meg. (csak számok esetén használható)
- **ReadOnly**: a jellemző igaz értéke esetén a mező nem szerkeszthető
- **Visible**: ezzel a jellemzővel a mezőt láthatatlanná tehetjük

A következő két jellemző (sajnos) csak a Delphi 3-ban található meg.

- **CustomConstraint**: egy mezőszintű ellenőrzést megvalósító feltétel
  Például ha egy személy nemét (*Nem*) 1 karakteren tároljuk, akkor a lehetséges értékek: 'F' és 'N'. Ez esetben a *CustomConstraint := (Nem = 'F') Or (Nem = 'N')*. A *MinValue* és *MaxValue* jellemzőkkel ezt a feltételt nem tudtuk volna beépíteni.
- **ConstraintErrorMessage**: ha a *CustomConstraint* jellemzőbe írt feltétel nem teljesül be, akkor a *ConstraintErrorMessage* jellemzőbe írt hibaüzenet fog megjelenni.
  Például visszatérve az előző példára, itt megadhatjuk, hogy milyen hibaüzenet jelenjen meg, ha a felhasználó nem megfelelő *Nem* értéket írna be:
  ConstraintErrorMessage := 'A személy neme csak "F" vagy "N" lehet.'

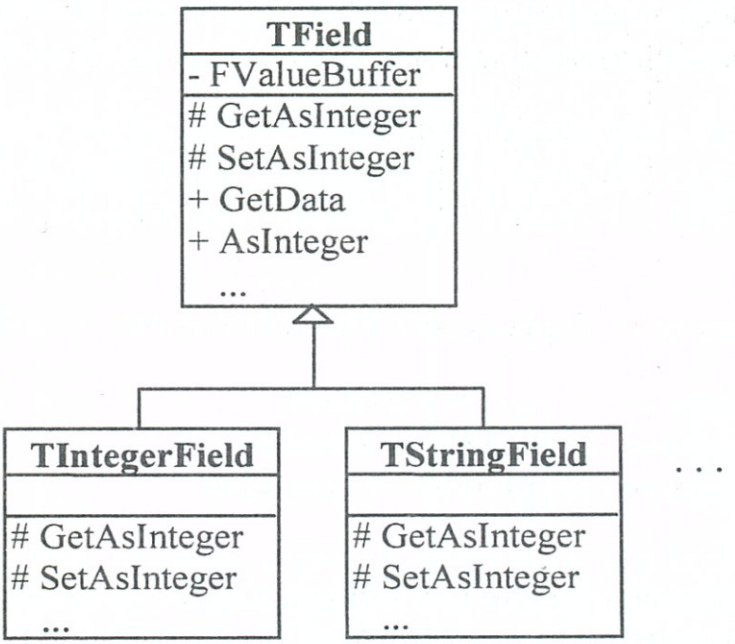
- Fontosabb eseményjellemzője:
  - **OnValidate**: segítségével mezőszintű ellenőrzést valósíthatunk meg. A mi ellenőrzésünk ily módon még az adatbázisba való lementés előtt fog lefutni. Ezt használhatjuk Delphi 1 és 2-ben a *CustomConstraint* helyett.

```
procedure TForm1.tblSzemelyValidate(Sender: TField);
begin
 if Not ((Sender.Value = 'F') or (Sender.value ='N')) Then
   begin
    ShowMessage('A személy neme csak ''F'' vagy ''N'' lehet.');
    Abort;
   end;
end;
```

A mezők konverziós jellemzői

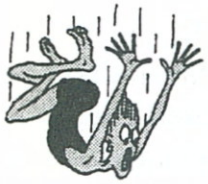
Ismerkedjünk meg a konverziós jellemzők implementálási hátterével!
Minden mező értéke egy *TField* osztálybeli privát mezőben tárolódik, az *FValuePuffer: Pointer*-ben (ez tulajdonképpen egy mutató a tényleges adatterületre). Ezt az értéket kellene a konverziós jellemzőknek látniuk, és természetesen átalakítaniuk a megfelelő típusra: az *AsInteger* alakítsa át *Integer*-ré, az *AsString String*-gé... Mivel jellemzőkről van szó, mindegyikük mögé elbujtathatunk egy *Set* és egy *Get* metódust. Például a *GetAsInteger* fogja a tényleges adatot, átalakítja egész számmá, majd eredményként ezt adja vissza. A *SetAsInteger* pedig a paraméterként átvett egész számot visszaalakítja a tényleges adat típusára, és úgy tárolja.

```
type
 TField = class(TComponent)
   private
     FValueBuffer: Pointer;
     ...
   protected
     function GetAsInteger:LongInt; virtual;
     procedure SetAsInteger(Value:LongInt); virtual;
     ...
   public
     property AsInteger:LongInt read GetAsInteger
                                write SetAsInteger;
     ...
 end;
```



8.16. ábra. A mezőobjektumok osztályhierarchia-diagramja

A konverzió algoritmusa erősen függ attól, hogy az adat eredetileg milyen formában található. Ebből azt a következtetést vonhatjuk le, hogy a konverziót megvalósító *Set* és *Get* metódusokat a *TField*-ből származó osztályokban köte-



lező módon felül kell írni. A *TStringField* osztályban maga az adat karakterlánc típusú, így itt a *GetAsInteger* és *SetAsInteger* metódusoknak a *String↔Integer* átalakításokat kell elvégezniük. A *TIntegerField* osztályban viszont, ahol az adat már eleve egész szám, a *Get* és *Set* metódusok egyenesen (átalakítás nélkül) szolgáltathatják az adatot. A *Set* és *Get* metódusok tehát a *TField* osztályban virtuálisak és védett (*protected*) láthatóságúak.

Már csak egy megválaszolatlan kérdés maradt: hogyan érik el az utód osztályok metódusai a tényleges adatot, hiszen ez privátként volt deklarálva a *TField* osztályban? A válasz egyszerű: van a *TField* osztályban egy nyilvános *GetData* függvény. Ezt hívják a felülírt *Get* és *Set* metódusok, így tudják a tényleges adatot elérni, állítani.

További implementációs részletek a SOURCE\VCL\DB.PAS állományban találhatók.

8.5.4 Hivatkozás egy adathalmaz mezőire

Egy adathalmaz mezőit a megfelelő mezőobjektumok segítségével lehet elérni. Ezek a mezőobjektumok vagy perzisztensek, vagy dinamikusak. A perzisztens mezőobjektumokat mi hozzuk létre tervezési időben, így mi is nevezzük el ezeket. Ez azt jelenti, hogy majd a nevük segítségével hivatkozni tudunk az értékeikre. A dinamikus mezőobjektumokat a rendszer hozza létre futási időben, ekkor nem tudjuk, hogyan nevezi el ezeket.
Továbbá, minden – perzisztens vagy dinamikus – mező értékét kiolvashatjuk és állíthatjuk mint *Variant* értéket, vagy típuskonverziók segítségével mint *Integer*, *String*, *Real*... értéket.
Foglaljuk össze, hogy mikor és hogyan lehet egy mezőre hivatkozni:

- Az adathalmazbeli mezőnév alapján (perzisztens és dinamikus mezőknél is):

```
Adathalmaz.FieldByName('MezőNeve').Value ⇨ Variant
Adathalmaz.FieldByName('MezőNeve').AsString ⇨ String
Adathalmaz.FieldByName('MezőNeve').AsInteger ⇨ Integer
...
```

- Mezőobjektum neve alapján (csak a perzisztens mezőknél):

```
MezőobjektumNeve.Value ⇨ Variant
MezőobjektumNeve.AsString ⇨ String
MezőobjektumNeve.AsInteger ⇨ Integer
...
```

- Mező adathalmazbeli indexe (sorszáma[1]) alapján (perzisztens és dinamikus mezőknél is). Egy adathalmazban az első mezőnek 0 a sorszáma, a másodiknak 1 stb.:

[1] Figyelem! Ne tévesszük össze egy mező indexét a tábla indexeivel: a mező indexe a táblabeli sorszámára vonatkozik, a tábla indexei pedig azok a segéd-adatszerkezetek, amelyek szerint az adathalmazt rendeztük a keresések gyorsaságának érdekében.

```
Adathalmaz.Fields[MezőIndexe].Value ⇨ Variant
Adathalmaz.Fields[MezőIndexe].AsString ⇨ String
Adathalmaz.Fields[MezőIndexe].AsInteger ⇨ Integer
...
```

Ez a hivatkozási mód akkor rendkívül hasznos, amikor egy ciklusban szeretnénk az összes mezőt sorban feldolgozni (például programból be akarjuk állítani minden mező szélességét, vagy ki akarjuk olvasni egy adathalmaz mezőinek elnevezését).

Mező	Mezőobjektum	Index (Sorszám)
OrderNo	tblItemsOrderNo:TFloatField	0
ItemNo	tblItemsItemNo:TFloatField	1
PartNo	tblItemsPartNo:TFloatField	2
Qty	tTblItemsQty:TIntegerField	3
Discount	tblItemsDiscount:TFloatField	4
Description	tblItemsDescription:TStringField	5
ItemValue	tblItemsItemValue:TCurrencyField	6

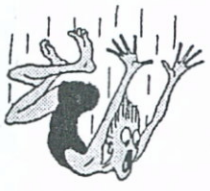
DM
(UDM.PAS)

```
uses UDM;
{Az alábbi utasítások egyenértékűek}
DM.tblItems.FieldByName('Qty').Value :=120;
DM.tblItems.FieldByName('Qty').AsInteger:= 120;

DM.tblItemsQty.Value:= 120;
DM.tblItemsQty.AsString:='120';

DM.tblItems.Fields[3].Value:=120;
DM.tblItems.Fields[3].AsFloat:=120;

{egyéb példák:}
DM.tblItems.FieldByName('Qty').MinValue := 1;
DM.tblItems.FieldByName('Qty').MaxValue := 1000;
DM.tblItems.FieldByName('Qty').DisplayLabel:='Mennyiség';
...
```

8.17. ábra. Hogyan hivatkozzunk egy adathalmaz mezőire?

8.6 A TTable komponens

- Helye az osztályhierarchiában: *TObject/TComponent/TDataSet/...TDBDataSet/TTable*
- Szerepe: biztosítja egy fizikai tábla vagy nézet (*view*) adatainak elérését. Ha adatainkat több táblából szeretnénk összeválogatni, akkor a *TQuery* komponenst használjuk (lásd 11. fejezet); ha pedig az adatok egy tárolt eljárás végrehajtásából keletkeznek, akkor ezek a *TStoredProc* komponens segítségével érhetők el Delphiből (lásd 14. fejezet).
- Jellemzői: a *TTable* komponensre tulajdonképpen minden érvényes, amit az adathalmazokról tanulmányoztunk: a megnyitási és bezárási módoktól, a lehetséges állapotain, a keresési és szűrési lehetőségeken keresztül, egészen a mezőkig. A továbbiakban koncentráljunk a *TTable* leggyakrabban használt jellemzőire:
  - **DatabaseName**: vagy egy adatbázis-komponens nevét tartalmazza, vagy egy létező álnevet. Az itt megjelölt adatbázisból származnak majd az adatok.
  - **TableName**: a lenyíló listából kiválogatjuk és beállítjuk a fizikai tábla vagy a nézet nevét
  - **Active**: lehetővé teszi a tábla megnyitását, így ennek adatai elérhetővé válnak
  - **ReadOnly**: Igazra állításával letilthatjuk a tábla adatainak módosítását
  - **IndexName**, **IndexFieldNames**: mindkét jellemző az adatok rendezettségi sorrendjének beállítására való. A használatban ezek a jellemzők kizárják egymást: vagy egy létező index nevét (például *ByCountryCity*) adjuk meg az *IndexName* jellemző segítségével, vagy az *IndexFieldNames* jellemzőben soroljuk fel az indexben szereplő mezőket ';'-vel elválasztva (például *Country;City*). Paradox és dBase tábláknál itt csak létező indexekre hivatkozhatunk. Adatbázisszerverek esetén akármilyen mező(k) szerint rendezhetjük adatainkat; ha a beállított mező(k) alapján nem létezik index, akkor a szerver felépít egy ideiglenest.
  - **MasterSource, MasterFields**: a fő-segéd űrlapok létrehozásánál használjuk. Bővebben lásd a 8.8. pontban.

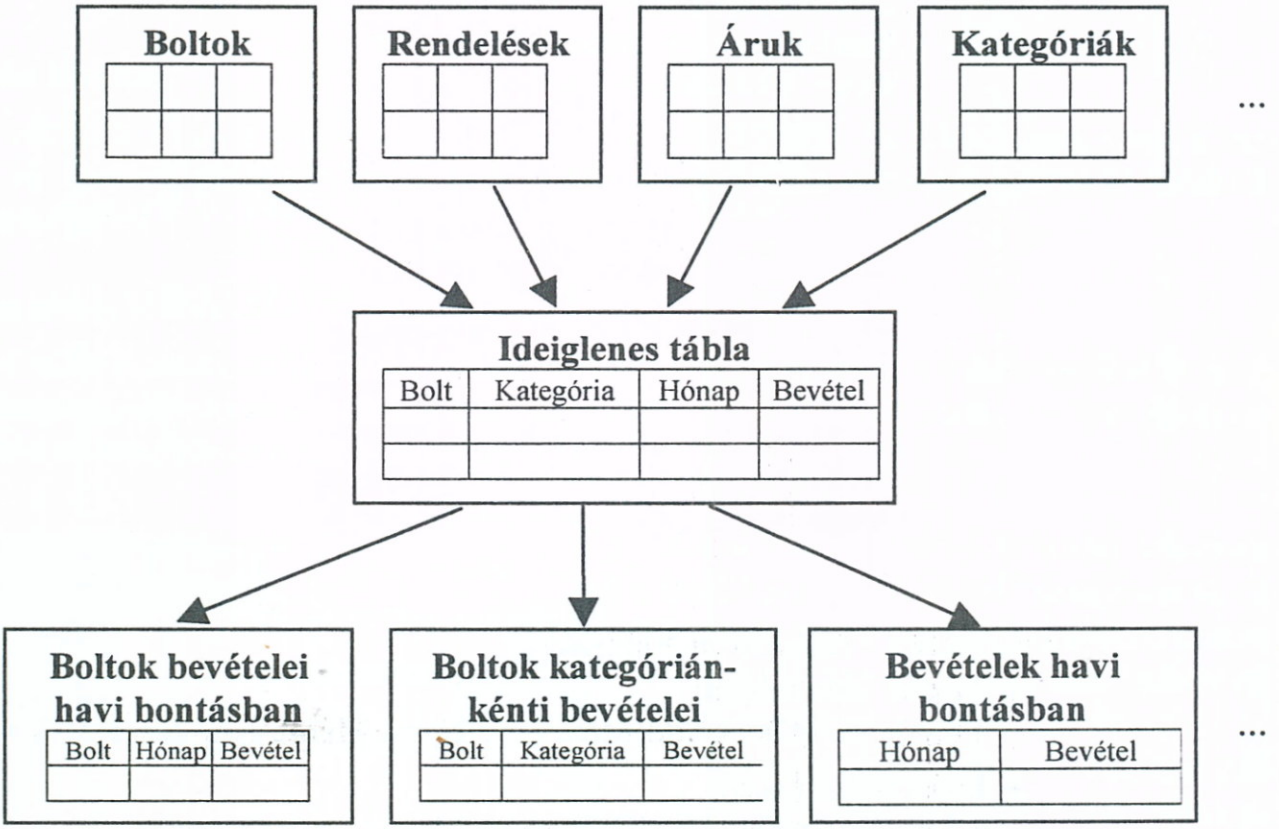


Tábla létrehozása programkódból

Adatbázisos alkalmazásainkban előfordulhat, hogy újabb táblákat kell létrehoznunk az alkalmazás futása közben. Tesszük ezt azért, mert például nem terveztük meg tábláinkat előtte (ez ritkán fordul elő), vagy mert ideiglenes eredményeket szeretnénk egy új táblában tárolni.

Tekintsünk át egy példát: egy bolthálózat nyilvántartójában az év végi statisztikai kimutatások (*boltonként a bevételek havi bontásban, boltonként és árukategóriánként a bevételek, havonként a bevételek, árukategóriánként a bevételek stb.*) mind lekérdezéssel oldhatók meg. A törzsadatok iszonyatos mennyisége viszont azt sugallja, hogy ezek lekérdezésének számát próbáljuk a minimálisra csökkenteni. Hatékonysági meggondolásokból készíthetünk egy ideiglenes táblát, melyben boltonként, árukategóriánként szerepelnek az összesített bevételek havi bontásban. Ennek az ideiglenes táblának további lekérdezésével fogjuk a

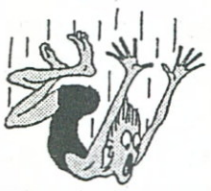
különböző statisztikai kimutatásokat megvalósítani. És mivel ez lényegesen kevesebb rekordot tartalmaz, mint a törzsadatok, a lekérdezések összességében nézve gyorsabban fognak eredményt szolgáltatni, mintha mindegyikük a törzsadatokat kérdezné le.
Tehát ha a statisztikai kimutatások lekérdezéseinek közös „gyökerük" van (és ez nem a törzsadat), akkor ajánlatos ezt egyszer lekérdezni, ideiglenes táblába tenni, és ezt dolgozni fel a továbbiakban. Ezt mutatja a 8.18. ábra is.



8.18. ábra Példa az ideiglenes táblák használatára

Táblát kétféleképpen hozhatunk létre: az első megoldás az, hogy lefuttatunk egy *CREATE TABLE* lekérdezést, majd miután már fizikailag megvan a tábla, létrehozunk, és ráirányítunk egy *TTable* komponenst; a másik megoldásban programból hozzuk létre a táblát, a *TTable* komponens segítségével. Nézzük mindkét megoldás megvalósítását: (A feladat szempontjából nem lenne kötelező az elsődleges kulcs definíciója, de a példa kedvéért hadd legyen.)

```
{tábla létrehozása SQL-lel; a konkrét szintaxis és adattípusok az adatbázis-formátumtól függnek; az alábbi utasítás
Paradox táblát hoz létre}
CREATE TABLE "Ideiglenes.DB"
     (BoltAz CHAR(5) ,
     KategoriaAz INTEGER,
     Honap INTEGER,
     Bevetel MONEY,
     PRIMARY KEY (BoltAz,KategoriaAz,Honap) )
```



```
{tábla létrehozása programkódból}
procedure TDM.TablaLetrehozas(Sender: TObject);
var tblUj:TTable;
begin
   tblUj:= TTable.Create(self);
   With tblUj Do
      begin
         DatabaseName:= 'DBDemos';
         TableName:= 'Ideiglenes.DB';
         With Fielddefs Do
            begin
               Add('BoltAz', ftString,5,True);
               Add('KategoriaAz',ftInteger,0,True);
               Add('Honap',ftInteger,0,True);
               Add('Bevetel',ftCurrency,0,True);
            end;
         IndexDefs.Add ('PrimaryIndex',
                        'BoltAz;KategoriaAz;Honap',
                        [ixPrimary,ixUnique]);
         CreateTable;
      end;
end;
```

Elemezzük a használt metódusokat:

- Mezők létrehozása:
  TTable.FieldDefs.Add(Mezőnév, Típus, Méret, KitöltéseKötelező-e):
- Index definiálása:
  TTable.IndexDefs.Add(IndexNév,Mezők, IndexTípus)
- A tábla létrehozása az eddigi beállítások alapján:
  TTable.CreateTable

Egy tábla feltöltése egy lekérdezés eredményével

Csupán a bemutatott példa teljességének kedvéért nézzük, hogyan lehet a létrehozott táblát egy lekérdezés eredményével feltölteni. Újból két megoldás közül kell választanunk: az első az, hogy a lekérdezésen végiglépkedve feltöltjük programkódból a táblát. Ez a lassúbb, kevésbé hatékony megoldás. A másik ötlet az *INSERT INTO* SQL utasítás használata. Körülbelül így:

```
INSERT INTO Ideiglenes
SELECT Boltok.BoltAz, Kategoriak.KategoriaAz...
FROM Boltok, Kategoriak...
WHERE...
```

Egyes adatbázis-szervereknél létezik beépített „ideiglenes tábla szolgáltatás". Ez azt jelenti, hogy lefuttathatjuk egyenesen a *SELECT INTO* SQL utasítást, anélkül, hogy előtte a táblát létrehoztuk volna. (MS SQL esetén például az SQL

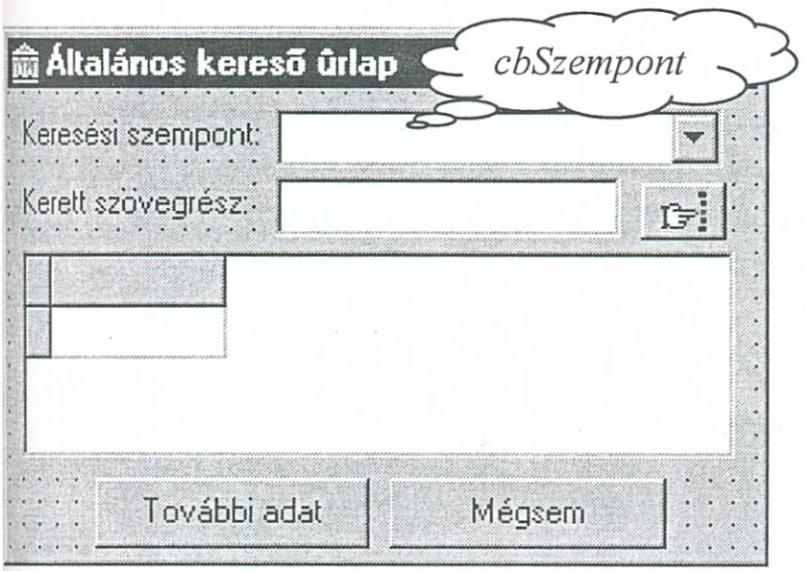
utasításban a tábla neve elé egy '#'-ot kell elhelyezni, ezzel jelezvén, hogy ideiglenes tábláról van szó. Pl.: *SELECT mezők INTO #Ideiglenes FROM Boltok,...*) Ekkor a rendszer létrehozza ezt, és feltölti adatokkal (az utasítás SELECT részének eredményével). Ezek után úgy dolgozhatunk vele, mint egy akármilyen másik táblával. A munka végeztével (az adatbázis-szerverből való kijelentkezéskor) a szerver automatikusan megszünteti az összes ideiglenes táblát.

Létező tábla törlése

Egy létező táblát a *TTable.DeleteTable* metódussal törölhetünk le. Ekkor nem csak a tartalma vész el, hanem maga a tábla is. SQL–lel is megtehetjük mindezt, a DROP TABLE utasítás segítségével.

Tábla indexeinek leolvasása programkódból

Adatbázisos alkalmazásainkban több táblában több szempont szerinti keresgélést is meg kell valósítanunk. Ilyenkor tervezhetünk egy általános kereső-űrlapot, melyet minden egyes keresés esetén feltöltünk a konkrét adatokkal, majd megjelenítünk.



8.19. ábra. Általános kereső-űrlap

Az általános kereső űrlapon lehessen beállítani a keresési szempontot, a keresett szöveget, majd keresés után a találatról további adatokat lehessen megjeleníteni.

A keresési szempontot a felhasználó majd egy kombinált listából választhatja ki: ha egy konkrét alkalmazottra kíváncsi, akkor beosztása, neve alapján tudjon rákeresni, ha viszont egy bizonyos árucikket akarna megtalálni, akkor árukategória és név alapján tudja ezt megtenni. Ha azt szeretnénk, hogy e szempontok szerint gyors legyen a keresés, akkor már a táblák tervezésénél létre kell hoznunk számukra az indexeket. A keresési szempont kombinált listát (*cbSzempont*) az űrlap megjelenítése előtt fel kell töltenünk a konkrét tábla indexelt mezőivel. Egy tábla indexeit az ***IndexDefs*** jellemzőjével kérdezhetjük le. Az ***IndexDefs.Count*** megadja a tábla indexeinek számát, a konkrét indexelt mezőket pedig az ***IndexDefs.Items[i].Fields***-szel kérdezhetjük le. Ha az index több mezőből áll, akkor ezek nevei ';'-vel vannak elválasztva.

Íme a kód:

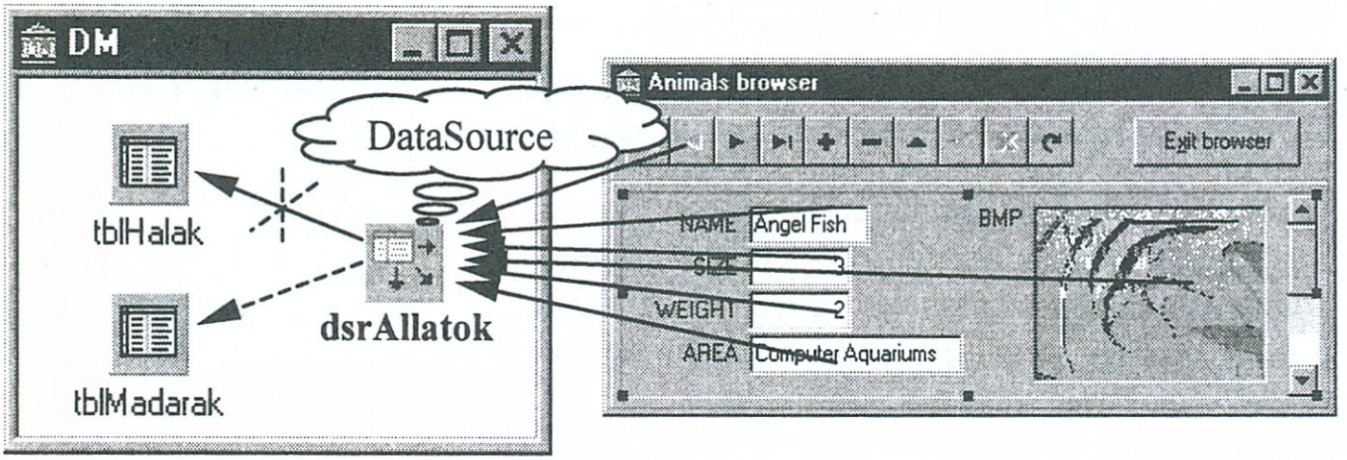
```
procedure TfrmMain.AlkalmazottKeresesClick(Sender: TObject);
var i:Integer;
begin
   With frmAltalanosKereso Do
      begin
         cbSzempont.Items.Clear;
         DM.tblAlkalmazott.IndexDefs.Update;
         For i:=0 to DM.tblAlkalmazott.IndexDefs.Count-1 Do
            cbSzempont.Items.Add(
                        DM.tblAlkalmazott.IndexDefs.Items[i].Fields);
         cbSzempont.ItemIndex:=0;
         {további beállítások}
         if ShowModal= mrOk Then
            {további adat megjelenítése}
      end;
end;
```

Futtassa le a 8_INDEXLEOLVASAS\PINDEXEK.DPR alkalmazást! Segítségével élőben is kipróbálhatja az indexek leolvasását.

Íme megismerkedtünk *aTTable* adathalmaz-komponenssel. További ilyen komponensek a *TQuery* és a *TStoredProc*. Ez a fejezet már így is elég hosszúra sikeredett, ezért ezen komponensek bemutatását későbbi fejezetekre halasztjuk: a *TQuery*-ről a 11., a *TStoredProc*-ról pedig a 14. fejezetben lesz szó.

8.7 A TDataSource komponens

- Helye az osztályhierarchiában: *TObject/TComponent/TDataSource*
- Szerepe: az adatok megjelenítését teszi lehetővé úgy, hogy a különböző adatelérési komponenseket összekapcsolja az adatmegjelenítési komponensekkel. Ha tehát egy adathalmaz tartalmát meg szeretnénk jeleníteni egy űrlapon, akkor rá kell irányítanunk egy *DataSource* komponenst, a megjelenítési elemek pedig ehhez csatlakoznak. Hogy miért ez a közbülső szint? Ennek egyik okát mutatja a 8.20. ábra: ha meg szeretnénk jeleníteni több, azonos szerkezetű adathalmaz tartalmát, akkor nem kell mindegyiküknek külön űrlapot terveznünk. Meg tudjuk ezt oldani egy űrlappal is, úgy, hogy az adatforrást, amiből az űrlapon levő megjelenítési elemek táplálkoznak, átirányítjuk egy másik, hasonló adathalmazra. Így **egyetlen** utasítással megoldhatjuk a helyzetet, nem kell minden egyes megjelenítési elem adatforrását átírogatnunk.



8.20. ábra. Az adatforrás (*TDatasource*) egyik előnye

- Jellemzői:
  - **DataSet**: annak az adathalmaznak a neve, amelynek a tartalmát továbbítja a megjelenítési komponensek felé, és amelybe ír, amikor a felhasználó a megjelenítési elemekben módosításokat végez.
  - **AutoEdit**: ha Igaz értékű (ez az alapértelmezés), akkor a kapcsolt adathalmaz automatikusan *dsEdit* állapotba kerül, valahányszor a felhasználó az adatmegjelenítési komponensekben szerkeszteni próbálja az adatot. Ezt legtöbb alkalmazásunkban le szoktuk tiltani, hiszen balesetszerű adatelrontásokhoz vezethet. Ha értéke Hamis, akkor a tábla (vagy egyéb adathalmaz) csak az *Edit* metódus explicit hívására kerül szerkeszthető állapotba; ez a felhasználó szemszögéből annyit jelent, hogy külön meg kell nyomnia egy gombot ahhoz, hogy szerkeszthesse az adatokat.
- Eseményjellemzői:
  - **OnStateChange**: a kapcsolt adathalmaz állapotváltásakor következik be. Ha például nem használjuk a navigátorsort, hanem saját gombokkal oldjuk meg a szerkesztést, postázást..., akkor gombjaink engedélyezésére és leszürkítésére kiválóan alkalmas az *OnStateChange* esemény.

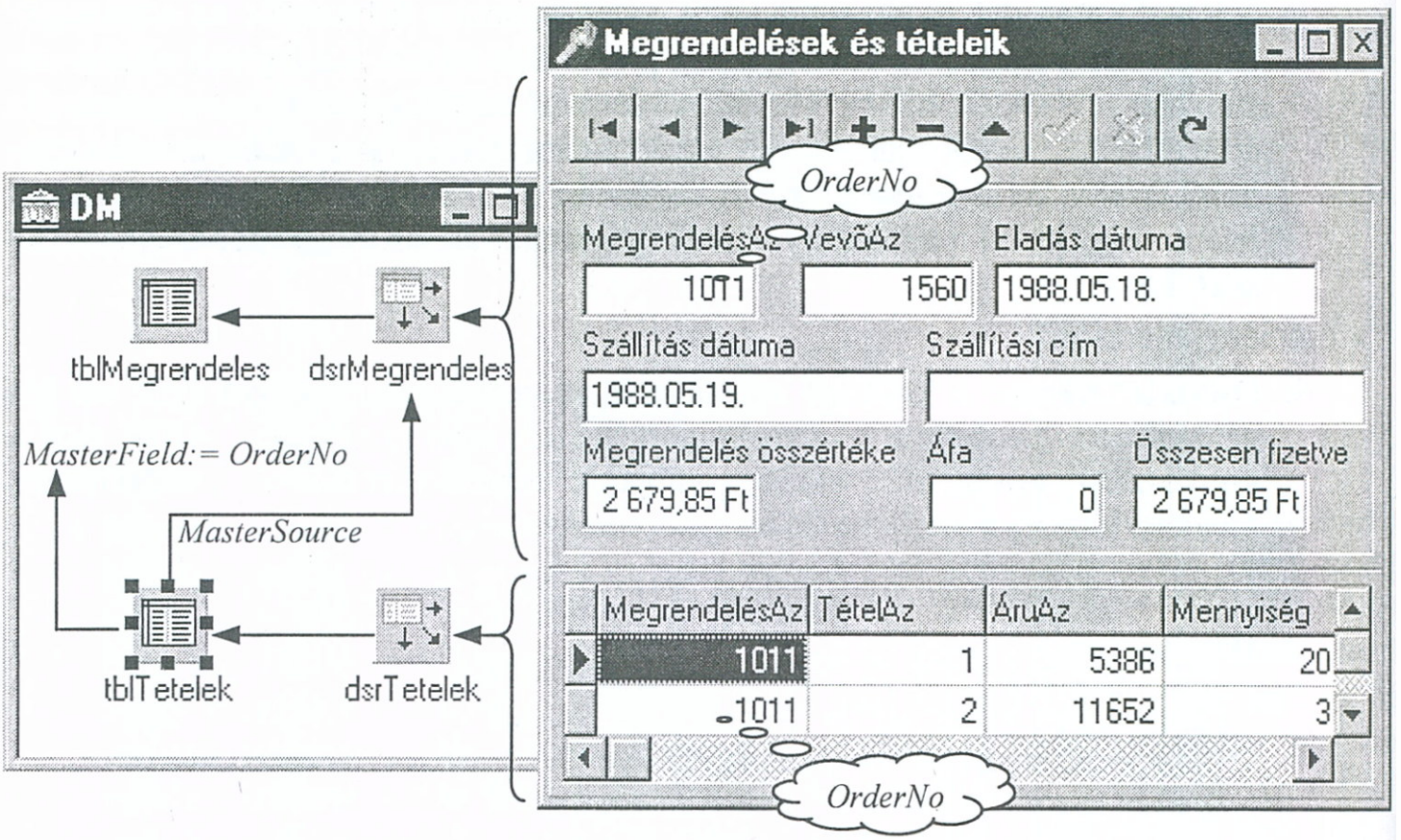
```
Procedure TfrmPelda.DataSource1StateChange(Sender: TObject);
Begin
   EditBtn.Enabled := Table1.State = dsBrowse;
   InsertBtn.Enabled := EditBtn.Enabled;
   PostBtn.Enabled := Table1.State in [dsInsert,dsEdit];
   CancelBtn.Enabled := PostBtn.Enabled;
   ...
End;
```

8.8 Fő-segéd űrlapok készítése

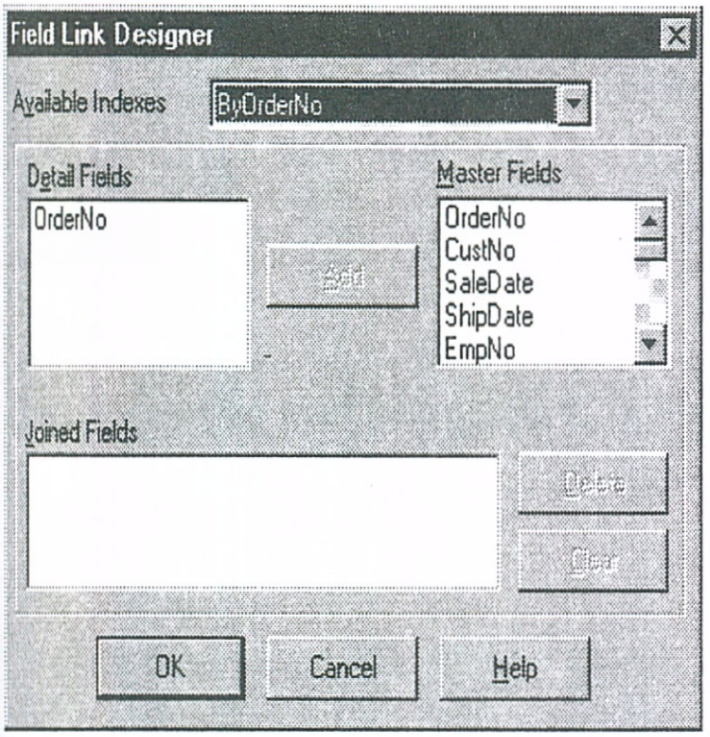
Fő-segéd űrlapnak nevezzük azt az űrlapot, amelyen legalább két, egy-a-többhöz kapcsolatban álló tábla tartalmát jelenítjük meg (lásd 8.21. ábra).

A felső részében egy megrendelés adatait láthatjuk (a *dsrMegrendeles*-ből), míg az alsóban (csak) **az aktuális** megrendelés tételeit (a *dsrTetelek*-ből). A *tblTetelek* komponens tartalmát tehát meg kell szűrnünk; mondhatjuk úgy is, hogy a *tblTetelek* táblát felülről, a *tblMegrendeles*-ből irányítják, vagyis, hogy a *tblItems* táblának van egy „mestere" – a *tblMegrendeles*.

Delphiben ezt a szűrést a táblakomponens *MasterSource* és *MasterFields* jellemzőivel valósíthatjuk meg. A szűrt tábla (*tblTetelek*) *MasterSource* jellemzőjét állítsuk be arra az adatforrásra, amelyik ezt irányítja (*dsrMegrendeles*), majd a *MasterFields* jellemzőjében mondjuk meg, hogy mely mezők valósítják meg a kapcsolatot.
Példánkban: *tblTetelek.OrderNo = tblMegrendeles.OrderNo*, azaz a tételek közül csak azokat szeretnénk egy adott pillanatban látni, amelyekben az *OrderNo* mező értéke megegyezik a megrendelések tábla aktuális rekordjának *OrderNo* értékével. A kapcsolódó mezők beállításánál használatos párbeszédablakot a 8.22. ábra mutatja be.



8.21. ábra. Fő-segéd űrlap megvalósítása



8.22. ábra. A kapcsolódó mezők specifikálása

A bal listadobozban (*Detail fields*) a tételek tábla aktuális indexmezőjét (mezőit) láthatjuk, míg a jobb oldaliban (*Master Fields*) a mestertábla (*Megrendelések*) mezőit. A feladat az, hogy a két tábla megfelelő mezőit összekapcsoljuk. A mi esetünkben a *tblTetelek. OrderNo* mező kapcsolódik a *tblMegrendeles.OrderNo* mezővel. A kapcsolás előtt a tételek táblában be kell állítanunk a kapcsoló mező (-k) szerinti indexet (*ByOrderNo*). A létező indexek az *Available Indexes* kombinált listában találhatók. Ha itt nem találjuk meg a keresett indexet, akkor lépjünk ki innen, hozzuk létre, majd térjünk vissza erre a pontra. Miután beállítottuk a megfelelő indexet, következhet a mezők összekapcsolása: jelöljük ki mindkét listában az *OrderNo* mezőt, majd kattintsunk az *Add* gombra. Hatására a *Joined Fields* dobozban megjelennek a mezők.

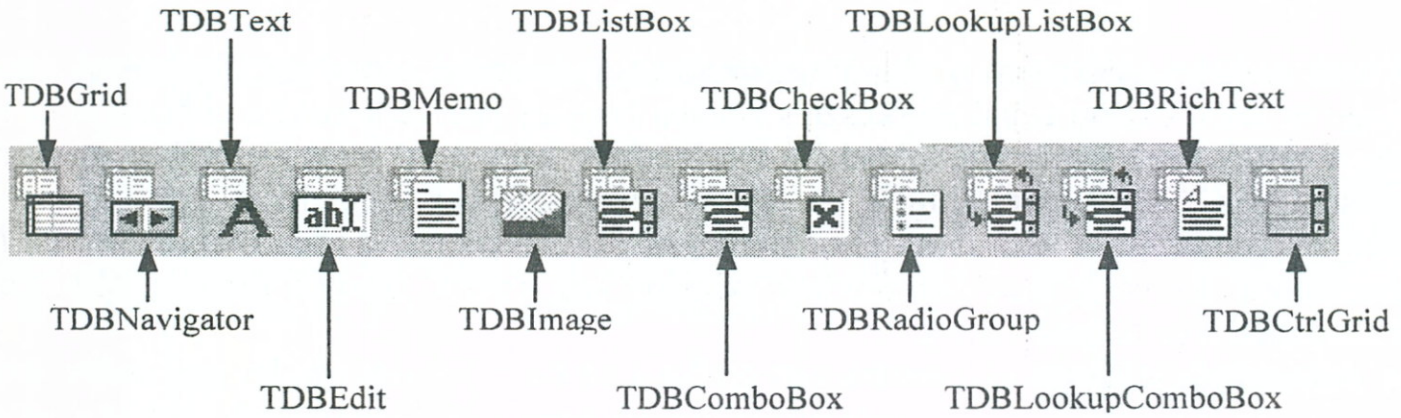
Ezzel a technikával tetszőleges mélységű fő-segéd űrlapokat készíthetünk. Például megjeleníthetjük az űrlap felső részén a vevőket, a középsőben az aktuális vevő megrendeléseit, az alsóban pedig az aktuális vevő aktuális megrendelésének tételeit. Csak arra kell vigyáznunk, hogy az egész még átlátható legyen. Több mint 3 szintes űrlapot csak nagyon indokolt esetben hozzunk létre.

Az itt bemutatott fő-segéd űrlapot a 8_FOSEGED\PMASTDET.DPR alkalmazásban találja. Próbálja ki, majd egészítse ki még egy szinttel: *Vevők (Customer)* ← *Megrendelések (Orders)* ← *Tételek (Items)*.

A következő fejezetben röviden bemutatjuk az adatmegjelenítési komponenseket, majd a 10. fejezetben begyakoroljuk eddigi adatbázisos ismereteinket.

9. Adatmegjelenítési komponensek

A *Data Controls* palettán elhelyezkedő komponenseknek interfész szerepük van: segítségükkel megjeleníthetjük és módosíthatjuk egy adatforrás tartalmát. Az adatmegjelenítési komponensek közül a legtöbbnek van nem-adatbázisos megfelelője is: a *TDBEdit* ↔ *TEdit, TDBText*↔*TDBLabel, TDBMemo*↔*TMemo, TDBCheckBox*↔ *TCheckBox*... mind párok (az ikonjaik is hasonlóak, csak az adatbázisos komponenseknél megjelenik a háttérben egy tábla). Más környezetekben (például Access, Visual Basic) nincsenek ennyire különválasztva az adatbázisos és nem-adatbázisos vezérlőelemek. Ott, ha például egy szerkesztődobozt „hozzákötünk" egy adatforráshoz, akkor ez adatbázisos lesz, ellenkező esetben pedig sima szerkesztődobozként viselkedik. Delphiben külön komponenseket vezettek be, de szerepükben, működésükben nem sok különbség van (azon kívül, hogy egyik adatbázisos, a másik meg nem). Emiatt itt csak az érdekesebbekről fogunk pár szót szólni.



9.1. ábra. Adatmegjelenítési komponensek

9.1 Az adatmegjelenítési komponensek használata

Az adatmegjelenítési komponensek használatának lépései:

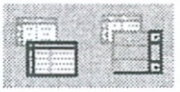
- Elhelyezzük az űrlapon.
- Beállítjuk **DataSource** jellemzőjét a megfelelő adatforrásra.
- Beállítjuk a **DataField** jellemzőjét a megfelelő mező nevére.

Például:

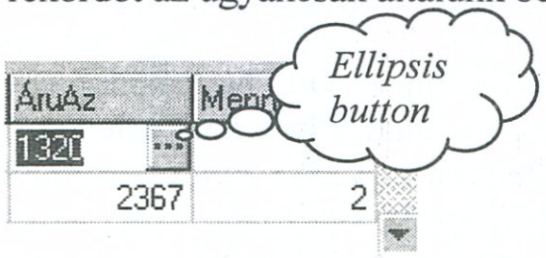
```
DBEdit1.DataSource := dsrSzemely;
DBEdit1.DataField := 'Nev';
DBGrid1.DAtaSource := dsrSzemely;
```

A *DBGrid, DBCtrlGrid* és *DBNavigator* az egész adatforrást kezelik, náluk nincs *DataField* beállítás.

- Egyes komponenseknél még szükség lehet további, egyéni beállításokra; ezeket a következő pontokban ismertetjük.

9.2 TDBGrid, TDBCtrlGrid 

E két rácskomponenssel egy adatforrás több rekordját is meg tudjuk egyszerre jeleníteni: a *DBGrid* egy rácsot biztosít erre a célra, a *DBCtrlGrid* pedig az általunk beállított számú rekordot az ugyancsak általunk beállított vezérlőelemekben jeleníti meg.

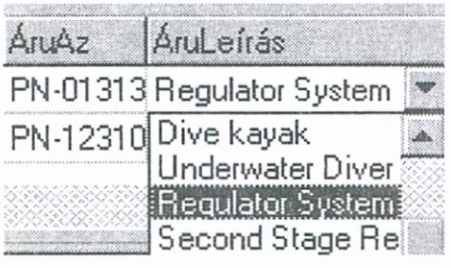


9.2. ábra. *DBGrid* komponens

A 16 bites Delphiben nincs *DBCtrlGrid*, csak *DBGrid*, és ez is szegényesebb, mint a 32 bites verziókban. Az újabb verziókban, ha duplán kattintunk a rácson, betöltődik egy rácsszerkesztő, mellyel testre szabhatjuk oszlopait. Beállíthatjuk, hogy mely mezőket szeretnénk látni, milyen szélességben, milyen színekben, mi legyen a fejlécben...

Külön említést érdemel az oszlopok ***ButtonStyle*** jellemzője. Ezzel azt határozhatjuk meg, hogy az adott oszlop cellái szerkesztés közben hogyan viselkedjenek: a *cbNone* érték esetén a cellák sima szerkesztődobozként viselkednek, mint a 16 bites Delphiben megismert rácsnál. Ha *cbEllipsis*-t állítunk be, akkor szerkesztéskor megjelenik egy kis „három pöttyös gomb" (lásd 9.2. ábra). Ha rákattintunk, bekövetkezik a rács ***OnEditButtonClick*** eseménye, melyben megjeleníthetünk például egy másik űrlapot (ahol a felhasználó kikeresheti a beállítandó árut).

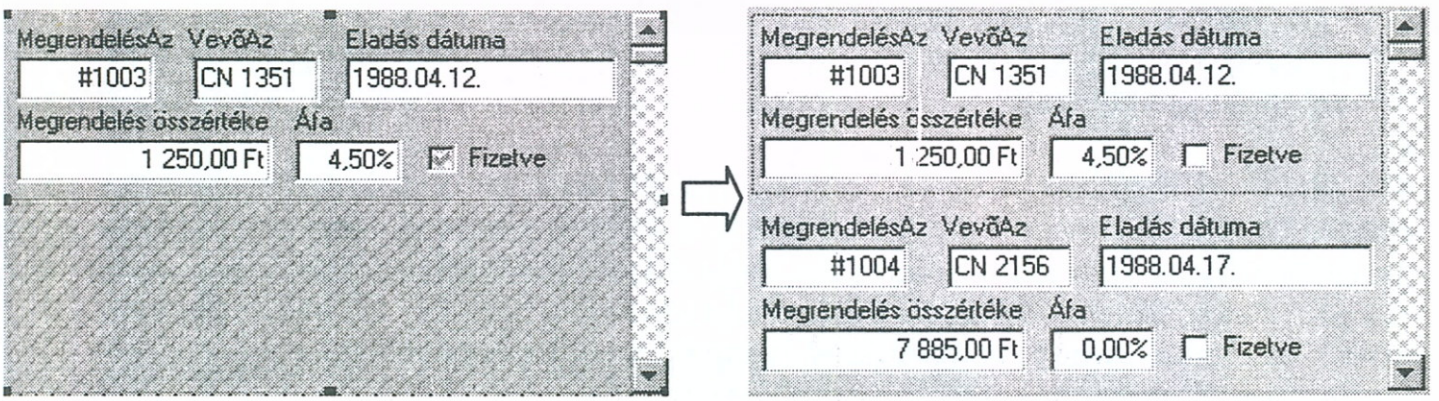
A *ButtonStyle* jellemző alapértelmezett értéke a *cbAuto*. Ez esetben, ha egy oszlop értéke valamilyen ismert értéklistából származik, akkor szerkesztéskor egy lebomló lista jelenik meg (lásd 9.3. ábra). Példánkban az *ÁruLeírás* egy „kikeresett" (*lookup*) mező, így a rendszer tudja, hogy a listát az *Áruk* tábla *ÁruLeírás* oszlopának értékeivel kell feltöltenie. (Ebben a megoldásban az a nagyszerű, hogy a származtatott mezőn keresztül módosítjuk az adatokat; ez olyan, mintha a származtatott mezőt szerkesztenénk.) Akkor is megjelenne a lebomló lista, ha az oszlop ***PickList*** jellemzőjét feltöltöttük volna a mező lehetséges értékeivel. Például, ha a személyekről nyilvántartjuk a nemüket is, a *Picklist* jellemzőnek a 'Férfi' és 'Nő' értékeket kellene tartalmaznia külön



9.3. ábra. DBGrid részlet

sorokban. Ez esetben beégetjük programunkba a lehetséges értékek halmazát (nem túl elegáns megoldás, de két, három **konstans** érték esetén fölösleges erre külön táblát felvennünk).
A *DBGrid* egy cellája tehát vagy sima szerkesztődobozként, vagy kombinált listaként viselkedik. Az *Ellipsis* gombbal rácsunkat tovább specializálhatjuk. Viszont a cellákban jelölőnégyzetet, vagy képet sehogyan sem tudunk megjeleníteni. Ha ezt szeretnénk, akkor a *DBCtrlGrid*-et használjuk.

A ***DBCtrlGrid***-nél megadhatjuk, hogy egyszerre hány rekordot akarunk látni (*RowCount*); a többit görgetéssel tudjuk elővarázsolni. Egy konkrét rekord külalakját mi határozzuk meg: a legfelső sorában elhelyezünk különböző adatmegjelenítési komponenseket (szerkesztődobozokat, jelölőnégyzeteket[1]...), beállítjuk ezek adatforrását és mezőjét. Ugyanezek fognak megjelenni a többi rekordban is, nyilván más és más adatokkal.



9.4. ábra. Egy *DBCtrlGrid* tervezési majd futási időben (Mi nem stimmel rajtuk?[2])

Mindkét rácstípusnál átfesthetjük a rekordokat (például a még ki nem fizetett megrendelések adatai piros háttérben jelenjenek meg). A *DBGrid*-nél az ***OnDrawColumnCell*** (régebbi verziókban ***OnDrawDataCell***) eseményjellemzőben kell az átfestést megvalósítanunk, a *DBCtrlGrid*-nél pedig a ***OnPaintPanel***-ben.

---

[1] A *DBCtrlGrid*-be elhelyezhető komponensek halmaza különböző a Delphi 2-es és 3-as verziókban. Például a 2-ben még nem lehetett benne képet (*DBImage*) megjeleníteni, a 3-ban ez már lehetséges. Egyik verzióban sem lehet viszont választógomb-csoportot tenni a *DCtrlGrid*-be.

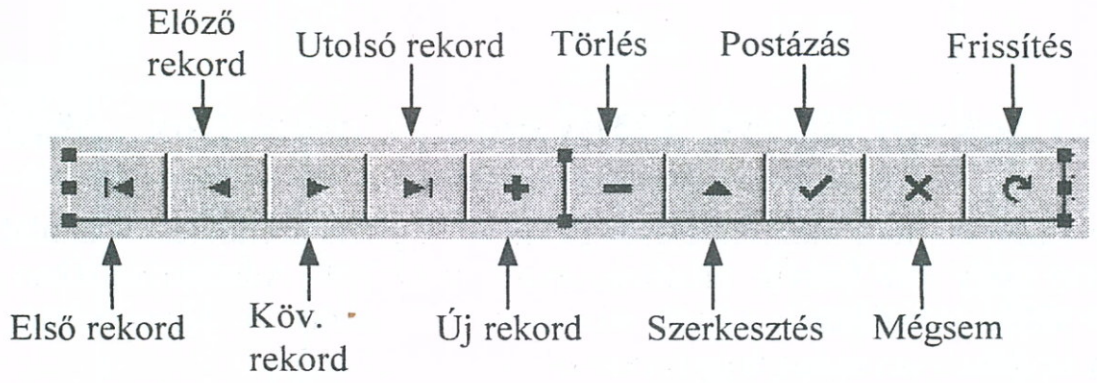
[2] Mindkét rácsban ugyanazt a rekordot látjuk, mégis az egyikben a *Fizetve* be van szürkítve, a másikban nincs. Ez azért van, mert a *Fizetve* számított mező, és emiatt tervezési időben még nem ismert az értéke. Így a rendszer inkább beszürkítette.

Íme a kód:

```
{DBGrid}
{Minden cellában beállítjuk a háttérszínt: pirosra, ha még nem fizet-
tek, egyébként pedig fehérre. Ha egy cella éppen fókuszban van, akkor
nem bántjuk, hadd „kéküljön" be.}
procedure TfrmRacsok.DbGrid1DrawColumnCell(Sender: TObject;
                const Rect: TRect;       {a megrajzolandó téglalap}
                DataCol: Integer;        {a megrajzolandó oszlop indexe}
                Column: TColumn;         {a rács oszlopai}
                State: TGridDrawState );{a kirajzolandó cella állapota}
begin
   If Not (gdFocused in State) Then      {ha a cella nincs fókuszban}
      If Not DM.tblMegrendelesFizetve.AsBoolean Then
         DBGrid1.Canvas.Brush.Color := RGB (255,200,200)
      Else
         DBGrid1.Canvas.Brush.Color := clWhite;
   grdMegrend.DefaultDrawColumnCell(Rect, DataCol, Column, State);
   {Ez a metódus írja ki a cellába az adatot}
end;

{DBCtrlGrid}
{A DBCtrlGrid minden egyes sora egy Panel-re kerül. Mi ennek a háttér-
színét festjük át. A cbFizetve jelölőnégyzetet is át kell festeni,
mivel ezt nem lehet átlátszóvá tenni, mint a címkéket.}
procedure TfrmRacsok.DBCtrlGrid1PaintPanel(DBCtrlGrid: TDBCtrlGrid;
                    Index: Integer);     {a kirajzolandó sor indexe}
begin
   With DBCtrlGrid Do
      begin
         If Not DM.tblMegrendelesFizetve.AsBoolean Then
            Canvas.Brush.Color:= RGB(255,200,200)
         Else
            Canvas.Brush.Color:=clSilver;
         cbFizetve.Color:=Canvas.Brush.Color;
         Canvas.Fillrect(Rect(0,0,PanelWidth, PanelHeight));
      end;
end;
```

Tanulmányozza a *DBGrd* és *DBCtrlGrid* komponensek tulajdonságait a 9_RACSOK\PRACSOK.DPR alkalmazásban. Jelenítse meg a rácsban a Vevő nevét is. Ha származtatott mezőként veszi fel, akkor a rácsban megjelenik majd a lebomló lista. Végül próbálja ki a rácsok formázási lehetőségeit: a fejlécben változtasson betűtípust és méretet, majd fesse át a *Kedvezmény* oszlopot a kedvenc színével.

9.3 TDBNavigator



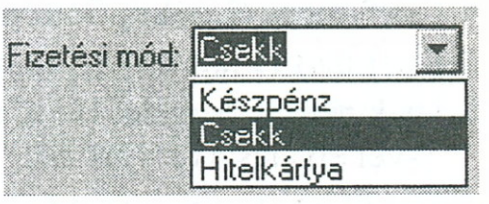
9.5. ábra. A navigátorsor gombjai és szerepük

A *Frissítés* gombnak hálózatos környezetben van értelme, ahol egy felhasználó megváltoztathatja „alattunk" az adatokat. Ilyenkor a *Frissítés* gombbal újraolvashatjuk az adattárat.
A *VisibleButtons* jellemzőjével az ábrán látható gombok közül akármelyik letiltható. Ha saját gombokkal szeretnénk megvalósítani a felvitelt, törlést, mentést és visszavonást, akkor ezeket a gombokat tüntessük el.

9.4 TDBListBox, TDBComboBox

Ezek tulajdonképpen a már ismert *TListBox* és *TComboBox* komponensek adatbázisos változatai. A *TDBListBox* és *TDBComboBox* komponensekben egy adott adatforrás egy adott mezőjének értékét jeleníthetjük meg, illetve szerkeszthetjük át. A (lebomló) lista tartalmát akár tervezési, akár futási időben **nekünk** kell szolgáltatnunk.
Például tegyük fel, hogy egy megrendelésről tárolni szeretnénk a fizetési módot: készpénz, csekk vagy hitelkártya. Ezt az információt, a megrendelést felvevő űrlapon egy három elemet tartalmazó kombinált listában kérhetjük be (lásd 9.6. ábra). És mivel a kiválasztott adatnak a *Megrendelés* tábla *FizetesiMod* mezőjébe kell bekerülnie, *TDBComboBox*-ot kell használnunk. Legyen a neve *dbcbFizetesiMod*. A következő kódrészletben a szükséges beállításokat láthatjuk.

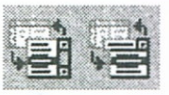
```
With dbcbFizetesiMod Do
  begin
    DataSource := DM.dsrMegrendeles;
    DataField:= 'FizetesiMod';
    With Items Do
      begin
        Add('Készpénz');
        Add('Csekk');
        Add('Hitelkártya');
      end;
  end;
```



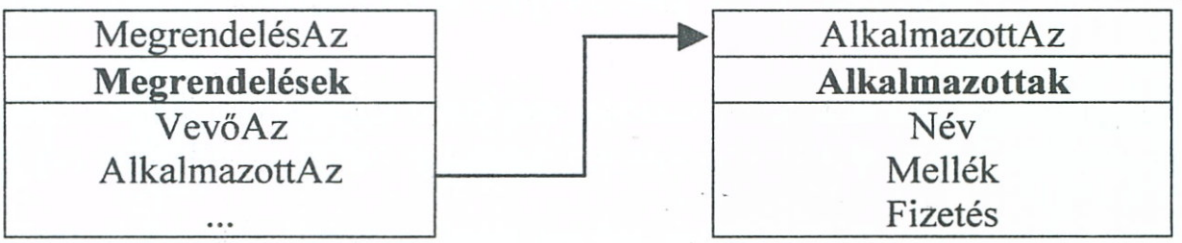
9.6. ábra. DBComboBox

Figyelem! Ha azt szeretnénk, hogy a lista tartalma automatikusan egy másik adatforrásból származzon, akkor a következő pontban tárgyalt *TDBLookupListBox* és *TDBLookupComboBox* komponenseket használjuk.

9.5 TDBLookupListBox, TDBLookupComboBox



Maradjunk az előbb említett megrendeléses példánál. Egy megrendelésről általában azt is tárolni szoktuk, hogy ezt melyik alkalmazottunk vette fel. Adatbázisunkban van egy *Alkalmazottak* tábla az alkalmazottaink adataival (9.7. ábra). A *Megrendelés* táblában természetesen az *AlkalmazottAz*-t fogjuk tárolni.



9.7. ábra. Egy alkalmazott több megrendelést is felvehet

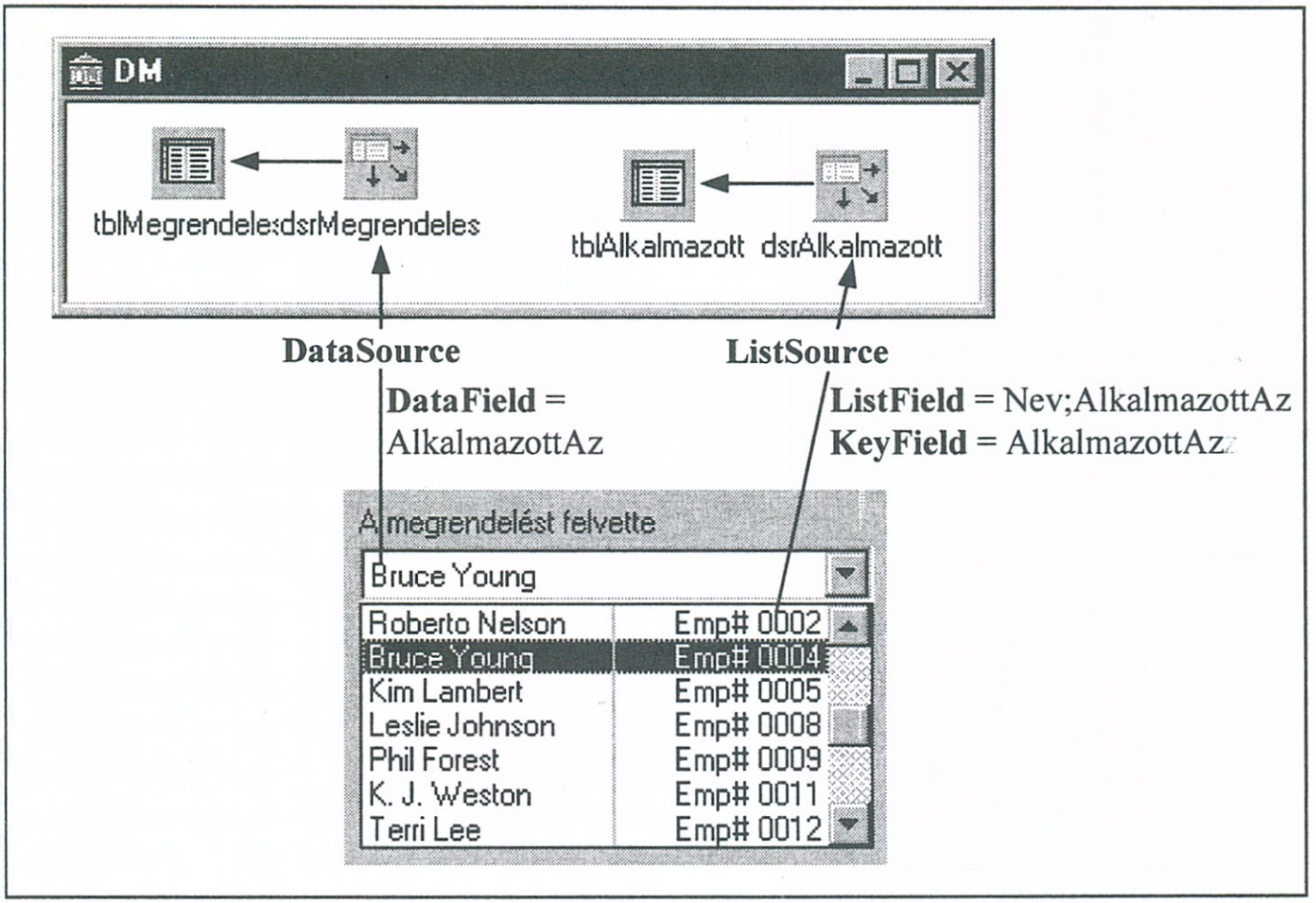
A megrendelést felvevő űrlapon az lenne az igazi, ha egy listából tudnánk kiválogatni a konkrét alkalmazottat. A *TBDListBox* és *TDBComboBox* komponensek itt nem használhatók, hiszen akkor nekünk kellene kézzel vagy programból felvinnünk az alkalmazottakat a listába. És mi történik akkor, ha felvesznek egy újabb alkalmazottat? Ő mikor kerülne be a listánkba?

Használjuk inkább a *TDBLookupListBox (TDBLookupList*[1]*)* vagy *TDBLookupComboBox (TDBLookupCombo)* komponenseket. Ezeknél a *ListSource*, *ListField* és *KeyField* jellemzők segítségével beállíthatjuk, hogy a lista tartalma honnan származzon, és mit tartalmazzon. (A jellemzőket a 16 bites Delphiben még másképp hívták, a régi elnevezésüket zárójelben tüntettük fel).

- **ListSource (LookupSource)**: melyik adatforrásból származik a lista tartalma.
- **ListField (LookupDisplay)**: a *ListSource*-ban megadott adatforrás mely mezői jelenjelek meg a listában. Ha több mező értékét is meg akarjuk jeleníteni, akkor a neveiket ';'-vel elválasztva írjuk be.
- **KeyField (LookupField)**: a listában kiválasztott rekord melyik mezőjének értékét írja be a *DataSource* adatforrás *DataField* mezőjébe.

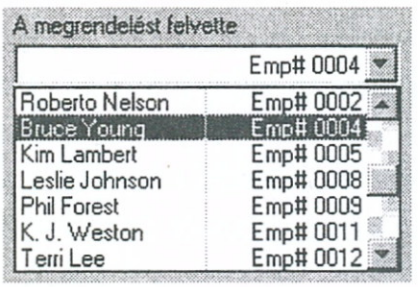
---

[1] Zárójelben a 16 bites Delphi-ben érvényes neveket tüntettük fel.



9.8. ábra. A *DBLookupComboBox* beállításai

A kombinált lista szerkeszthető részében a lista fölcsukódása után alapértelmezés szerint az első oszlopának értéke jelenik meg. A 16 bites Delphiben ezt nem is lehet megváltoztatni. Az újabb verziókban bevezettek egy *ListFieldIndex* nevű jellemzőt, melybe a megjelenítendő oszlop indexét kell beállítanunk. Ha tehát 0-ról átírjuk 1-re, akkor a következőt kapjuk:



9.9. ábra. Minden beállítás megegyezik az előzővel, csak a *ListFieldIndex*=1

A következő fejezetben begyakorolhatjuk eddigi adatbázisos ismereteinket egy konkrét példán keresztül.

10. Feladat: Könyvnyilvántartó

Az eddigiekben megismertük a Delphi adatbázis-kezelési alapjait: hogyan lehet adathalmazokat olvasni, írni és megjeleníteni. Lekérdezéseket és jelentéseket még nem tudunk Delphiben készíteni, ezzel később foglalkozunk (a lekérdezésekkel a 11., a jelentésekkel pedig a 13. fejezetben).
Apróbb feladatokat eddig is készítettünk. Most elérkezet annak az ideje, hogy egy bonyolultabb feladatot is összeállítsunk. Ebből a célból egy könyvtári nyilvántartó egy „szeletkéjét" fogjuk megírni, a könyveket és szerzőiket nyilvántartó részt. Ennek a kis feladatnak nem az a célja, hogy egy komplett rendszert bemutasson, hanem sokkal inkább az, hogy a „hogyan fogjak hozzá" kérdésre adjon választ, valamint az, hogy különböző felhasználói felület-tervezési ötleteket mutasson be. Kihasználva a Delphi vizuális űrlapörökítési mechanizmusát (lásd 5. fejezet), alkalmazásunk űrlapjainak egy hierarchiát fogunk kiépíteni, ezzel egy egységes és könnyen karbantartható felhasználói felületet biztosítva.
Ne gondoljunk túlságosan bele a példa valós életbeli vonzataiba, mindvégig a technikára fektetjük a hangsúlyt. A tárolandó adatokat is úgy válogattuk össze, hogy minél több Delphi-specifikus fogást lehessen rajtuk bemutatni.

Egyelőre még Paradox táblákkal dolgozunk, később egy Interbase (kliens/szerver) alapú adatbázis-kezelési feladatot is meg fogunk oldani (lásd 14. fejezet).

10.1 Feladatspecifikáció

Készítsünk egy könyvnyilvántartót, mely egy könyvtár könyveinek és ezek íróinak adatait kezeli. A könyveket témájuk szerint kategóriákba sorolva tartsuk nyilván. A programnak biztosítania kell az adatok karbantartását (felvitelét, módosítását, törlését), lehessen különböző szempontok szerinti kereséseket lebonyolítani, valamint adatainkat lehessen kilistázni.

Fogalmazzuk ezt meg pontosabban:
A program által nyilvántartandó adatok:

- Könyvekről:
  - ISBN szám
  - Cím
  - Szerzője (szerzői)
  - Kiadó
  - Témakör

- Leírás: néhány mondatos tartalma
- Példányszám (hány példányban szerepel a könyvtárban[1])
- Ára

- Írókról:
  - Név
  - Nem (csak a példa kedvéért)
  - Fénykép (csak a példa kedvéért)

A program funkciói:

- Könyvek nyilvántartása:
  - Karbantartás
  - Keresés: cím, ISBN és témakör szerint
  - Nyomtatás
- Írók nyilvántartása:
  - Karbantartás
  - Keresés név alapján
  - Nyomtatás
- Témák karbantartása és nyomtatása
- Kiadók karbantartása és nyomtatása

Ezt az alkalmazást 3 fejezeten keresztül fogjuk fejleszteni. A felsorolt funkciók közül a legtöbbet már ebben a fejezetben megoldjuk. A témakörök szerinti kereséshez lekérdezést fogunk írni, így ezt a 12. fejezetig elhalasztjuk. A nyomtatási lehetőségekkel is csak később, a 13. fejezetben ismerkedünk meg.

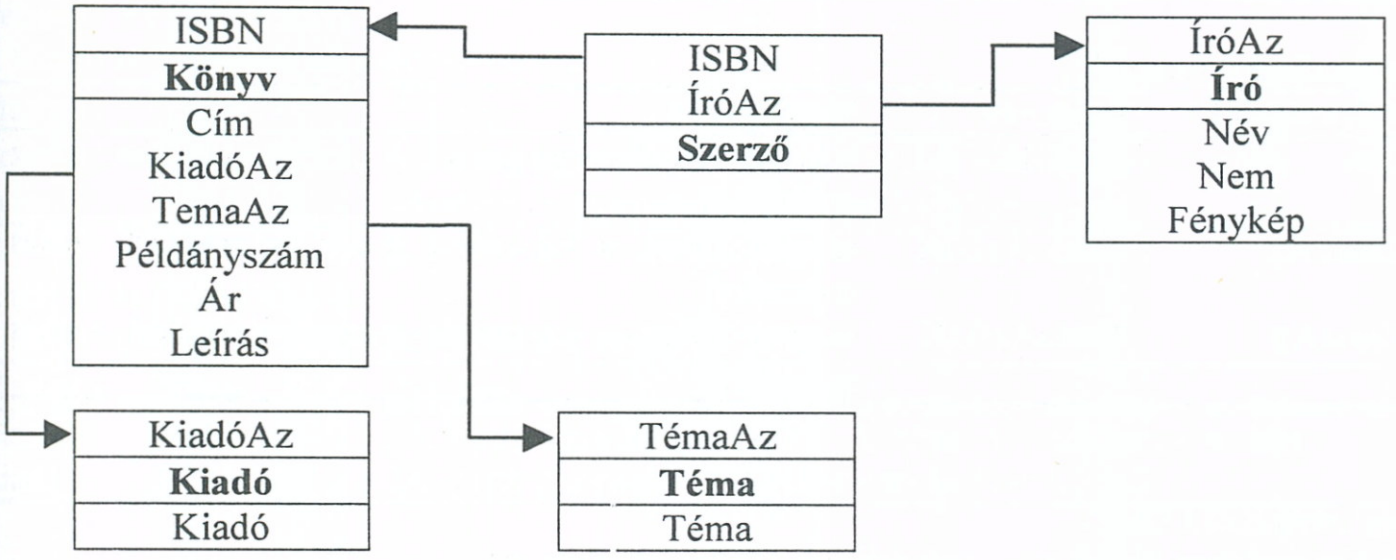
Megoldás (🖫 Alkalmazás: 10_KONYVTAR\PKONYVTAR.DPR
Adatok: ADATOK\KONYVTAR)

10.2 Az adatmodell

Az adatmodell elkészítése az adatbázisos feladatok fejlesztésének leglényegesebb pontja. Nagyon tömören megfogalmazva, az adatmodellnek kellően komplexnek kell lennie, ahhoz, hogy minden programfunkciót kielégítsen. Másrészről viszont vigyáznunk kell arra is, hogy fölösleges adatokkal ne bonyolítsuk rendszerünket. Csak a konkrét feladat szempontjából fontos adatokat kell tárolnunk, egy minél „egészségesebb" formában.
Az adatbázis-tervezés folyamatának ismertetése túlmutat könyvünk keretein, ezért itt csak a végeredményét, az implementálandó adatmodellt ismertetjük (10.1. ábra).

---

[1] Egy „igazi" könyvtári nyilvántartásban külön kellene tárolni a könyvek példányait, valamint a könyvtári tagokat is. A tagok nem a könyvet viszik el, hanem annak egy példányát... Itt az egyszerűség kedvéért mindezt kihagyjuk. Ne feledjük, a hangsúly a technikán van.



10.1. ábra. Az adatmodell

A *Könyv* és az *Író* táblák között több-a-többhöz kapcsolat van, így egy kapcsolótáblát vezettünk be (*Szerző*). Ebben adjuk meg, hogy melyik könyvet (*ISBN*), ki (esetleg kik) írták. E két mező együttesen alkotja az elsődleges kulcsot. A *Téma* és *Kiadó* szótárállományok, melyek egy-a-többhöz kapcsolatban állnak a Könyv táblával.

Nézzük a **táblák részletes leírását**. A mezőnevek ne tartalmazzanak ékezetes betűket.

Könyv tábla (KONYV.DB)
A könyvtárban szereplő könyvek adatait tárolja.

Mezőnév	Típus	Szerep
ISBN	Char(10)	A könyv egyedi azonosítója. Kötelező kitölteni. Formátuma: '000 00 0000 0'. A szóközöket nem tároljuk. Annak érdekében, hogy pontosan 10 számjegyet lehessen csak beütni, bemeneti maszkot fogunk definiálni.
Cim	Char(50)	A könyv címe. Kötelező kitölteni. Bármilyen karaktert tartalmazhat.
KiadoAz	Egész szám	A könyv kiadóját azonosítja. Kötelező kitölteni. Idegen kulcs szerepe van, ez a kapcsolómező a *Könyv* és *Kiadó* táblák között.
TemaAz	Egész szám	A könyv témakörét azonosítja. Kötelező kitölteni. Idegen kulcs szerepe van, ez a kapcsolómező a *Könyv* és *Téma* táblák között.
Peldanyszam	Egész szám	A könyv példányainak számát tartalmazza. Alapértelmezés szerinti értéke 1. Formátuma '0 db'.
Ar	Pénznem	A könyv egy példányának ára. Formátuma '0.00 Ft'. Nem kötelező kitölteni.
Leiras	Memo	A könyv rövid ismertetése. Szöveges mező, nem kötelező kitölteni. Sosem fogunk tartalma alapján keresni.

Másodlagos indexek

Indexmező(k)	Indexnév	Megjegyzés
Cim	CimIDX	Növekvő rendezettséget ad. A cím szerinti kereséshez szükséges.
KiadoAz	KiadoAzIDX	Idegen kulcs, a kapcsolat gyors bejárásáért szokás indexelni.
TemaAz	TemaAzIDX	Idegen kulcs, a kapcsolat gyors bejárásáért szokás indexelni.

Író tábla (IRO.DB)

A könyvtárban létező könyvek íróinak adatait tartalmazza.

Mezőnév	Típus	Szerep
IroAz	Számláló	Az író egyedi azonosítója. Értékét a rendszer generálja.
Nev	Char(30)	Az író neve. Kötelező kitölteni.
Nem	Logikai	Az író neme (csak a példa kedvéért vettük be). Lehetséges értékei: *True* ⇒ Férfi, *False* ⇒ Nő. Az űrlapokon természetesen a Férfi/Nő fog megjelenni.
Fenykep	Graphic (bináris formában tároljuk)	A szerző fényképe (csak a példa kedvéért vettük be).

Másodlagos indexek

Indexmező(k)	Indexnév	Megjegyzés
Nev	NevIDX	Név szerinti növekvő index.

Szerző tábla (SZERZO.DB)

Mezőnév	Típus	Szerep
ISBN	Char(10)	Idegen kulcs, a *Könyv* tábla *ISBN* mezőjéhez kapcsolódik.
IroAz	Egész szám	Idegen kulcs, az *Író* tábla *IroAz* mezőjéhez kapcsolódik.

Az ISBN és IroAz mezők együttesen alkotják a Szerző táblában az elsődleges kulcsot.

Másodlagos indexek

Indexmező(k)	Indexnév	Megjegyzés
IroAz,ISBN	IroAzISBNIDX	Az *IroAz* mező szerinti kereséshez szükséges növekvő index.

Téma tábla (TEMA.DB)
A könyvtári könyvek témaköreit tárolja.

Mezőnév	Típus	Szerep
TemaAz	Számláló	A téma egyedi azonosítója. Értékét a rendszer generálja.
Tema	Char(30)	A téma megnevezése. Kötelező kitölteni.

Kiadó tábla (KIADO.DB)
A könyvtári könyvek kiadóinak adatait tárolja.

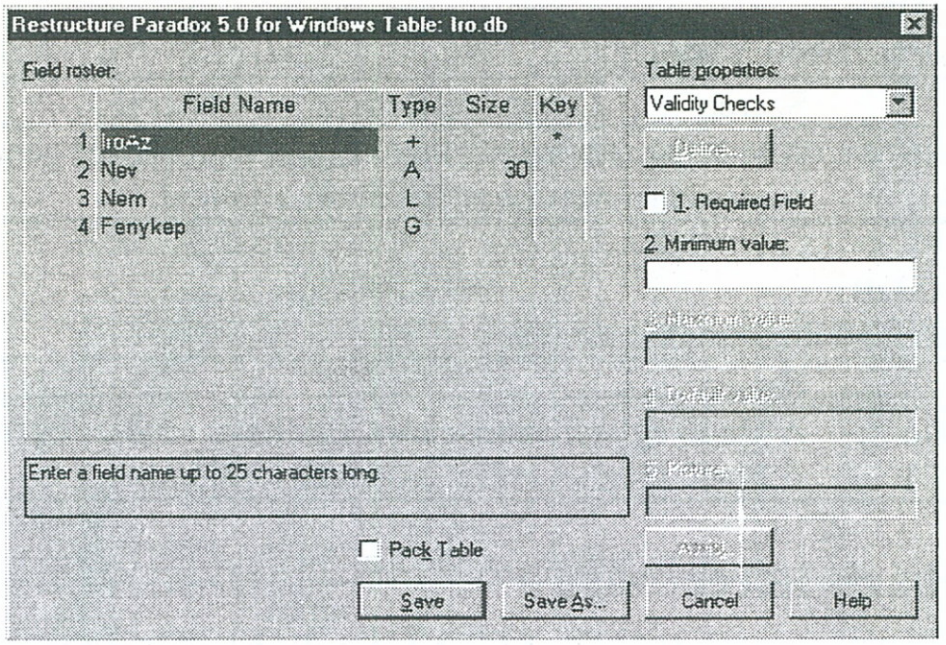
Mezőnév	Típus	Szerep
KiadoAz	Számláló	A kiadó egyedi azonosítója. Értékét a rendszer generálja.
Kiado	Char(30)	A kiadó megnevezése. Kötelező kitölteni.

10.3 Az adatbázis létrehozása

Következik a táblák fizikai megtervezése. Erre a célra a Database Desktop segédprogramot fogjuk használni. Indítása előtt azonban hozzunk létre egy új könyvtárat az adatállományok számára. Ezután hozzunk létre egy álnevet is (*DBKonyvtar*), mely erre a könyvtárra mutat. Az álnév készítését lásd a 7. fejezet 7.3. pontjában.

Ha ez megtörtént, akkor hajtsuk végre a következő lépéseket:

- Indítsuk el a Database Desktop segédprogramot.
- Állítsuk a munkakönyvtárat a leendő adataink könyvtárára. Ennek érdekében hívjuk meg a *File/Working Directory...* menüpontot, majd a megjelenő párbeszédablakban az *Aliases* kombinált listából válasszuk ki a frissen létrehozott álnevet. Ezután zárjuk be a párbeszédablakot.
- Készen állunk az első táblánk megtervezésére. De melyik legyen az? A táblák közötti kapcsolatokat mindig a többös oldalon levő táblánál kell megadni, ekkor pedig az egyes oldalinak már léteznie kell. Emiatt azt javaslom, hogy a többös oldali táblákat hagyjuk a legvégére. (Természetesen egy táblát később is meg lehet nyitni, és át lehet tervezni, így a kapcsolatot is megadhatnánk később; tábláink jó sorrendben való létrehozásával csupán időt spórolunk meg.) Legyen az első az *Író* tábla. Hívjuk meg a *File/New/Table...* menüpontot. Fogadjuk el a felkínált tábla formátumot (Paradox 7). Elemezzük a megjelent ablakot (10.2. ábra).
  - A *Field roster* részben sorba be kell gépelnünk a mezőket: nevüket, típusukat (ameddig nem tudjuk kívülről a típus rövidítését, addig a szökőz billentyűvel nyissuk le a listát, és onnan válasszuk ki a megfelelőt), méretét (csak a karakterláncnál kötelező). Az elsődleges kulcs mező(-k)nél az utolsó oszlopban duplakattintással, vagy a szóköz billentyű lenyomásával varázsoljunk egy csillagot. Ezt meg kell ismételni minden elsődleges kulcsot alkotó mezőnél (példánkban a *Szerző* táblában van kettős kulcs: *ISBN* és *IroAz*).

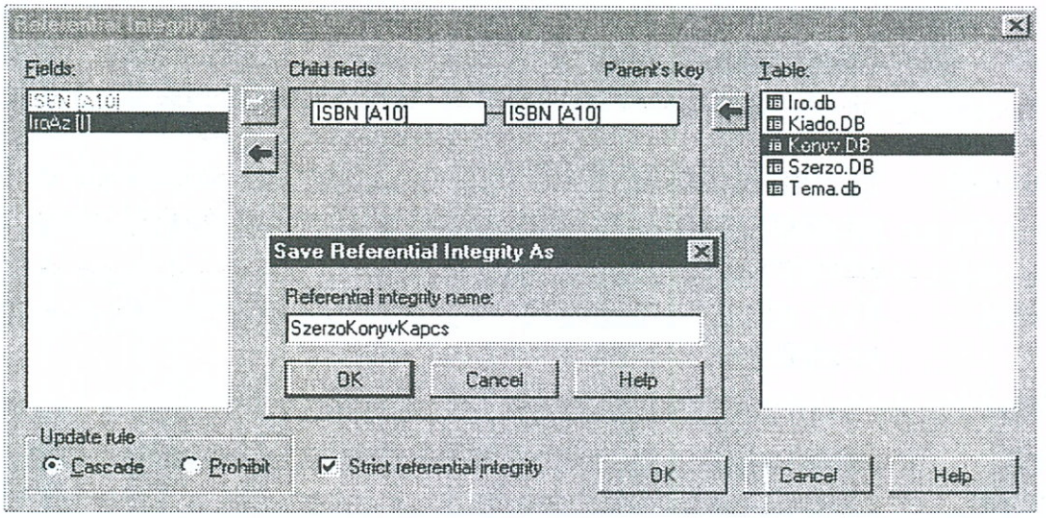


10.2. ábra. Táblák tervezése a Database Desktop-ban

- Az ablak jobb részében különböző táblaszintű és mezőszintű tulajdonságot állíthatunk be. Táblaszintűek: hivatkozási integritás (*Referential Integrity*), másodlagos indexek (*Secondary Indexes*), táblanyelv (*Table Language*)... Ne felejtsük el minden tábla esetében beállítani a Paradox HUN 852 kódlapot (10.3. ábra). Ennek hatására tábláink indexei a magyar ékezetes betűket a helyes sorrendben fogják tartalmazni.
  Mezőszintű beállítások: kötelező-e kitölteni (*Required Field*), minimális és maximális érték, alapérték (*Default Value*).

10.3. ábra. A táblanyelv beállítása

- Hozzuk létre sorban a táblákat a 10.2. pont táblázatai alapján. Készítsünk másodlagos indexeket is.
- Hivatkozási integritás beállítása: a *Szerző* táblának két hivatkozási integritási feltételnek is eleget kell tennie. Először be kell állítanunk az *Szerző-Könyv* kapcsolatot, majd a *Szerző-Író* kapcsolatot. A *Szerző-Könyv* kapcsolat megadását a 10.4. ábra is mutatja.



10.4. ábra. A hivatkozási integritás beállítása

- Az adattáblák megtervezése, indexek és hivatkozási integritások beállítása után, következhet a táblák tesztadatokkal való feltöltése. Figyelem, a hivatkozási integritás miatt előbb az egyes oldalon levő táblákat töltsük fel, és csak utána a többös oldalon levőket. Az írók fényképeit majd az alkalmazásunkból fogjuk feltölteni.

10.4 Az alkalmazás űrlapjainak megtervezése

Ennek a pontnak céljai a következő:

- A szükséges űrlapok külalaki és működési követelményeinek kidolgozása
- A szükséges jelentések megállapítása
- Az alkalmazás felhasználóbarát menüszerkezetének kialakítása, valamint az alkalmazás űrlapjai között egy logikus bejárási sorrend kigondolása.

Minden űrlap és jelentés esetén le kell rögzítenünk ennek szerepét, külalakját, és nem utolsósorban azt is, hogy mely adatokat írja, és melyeket olvassa.

Nagyobb alkalmazások esetében az űrlapok tervezését meg kell előznie egy komoly rendszerterv elkészítésének. A rendszerterv kidolgozása a szervezők feladata.

Mint erről már az 5. fejezetben szóltunk, Delphiben lehetőség van a vizuális űrlapöröklésre. A felhasználói felületet azért is nagyon fontos még az alkalmazás implementálásának a legelején kigondolnunk, mert így ki tudjuk választani a közös külalaki és működési elemeket, és ezáltal egy űrlaphierarchiát alakíthatunk ki alkalmazásunk számára.
Elemezzük a mi kis rendszerünk funkcióit és megvalósításukat. (Az aláhúzások értelmezését lásd később.)

Az adatbázis szerkezete nincs kihatással a felhasználói felület felépítésére. A felhasználónak minél barátságosabb formában kell „tálalnunk" az adatokat akkor is, ha azokat több táblából kell „összevadásznunk". Például csak azért, mert a mi esetünkben a szerzők tábla külön van, nem fogjuk a felhasználót is arra kényszeríteni, hogy ezt az információt egy külön űrlapon vigye föl.

Funkció	Megvalósítása
Könyvek:	
• Karbantartás	Fő-segéd űrlap: fölül egy navigátorsor segítségével lépegetünk és szerkesztjük a könyveket, alul pedig egy rácsban az aktuális könyv szerzőit szerkeszthetjük.
• Nyomtatás	Egy nyomtatás gomb segítségével kilistázzuk a könyveket és szerzőiket.
• Keresés:	
↳ Cím szerint	Begépeljük egy szerkesztődobozba a keresett címet, majd egy keresés gomb hatására indítjuk a keresést a *Könyvek* táblában.
↳ Témakör szerint	Egy kombinált listából kiválasztjuk a kívánt témakört. A hozzátartozó könyvek egy rácsban fognak megjelenni.
A könyvek karbantartását, nyomtatását és keresését egyetlen űrlapon fogjuk megvalósítani. A karbantartás és a keresés több helyet igényel, emiatt ezeket egy füzet (*PageControl*) különböző oldalaira helyezzük. A nyomtatás egy gombot feltételez csak, így ennek nem kell külön oldal.	
Írók:	
• Karbantartás	Fő-segéd űrlap: fölül egy navigátorsor segítségével lépegetünk és szerkesztjük az írók adatait, alul pedig egy rácsban jelenítsük meg az aktuális író könyveit. Itt a könyvek adatait ne lehessen szerkeszteni.
• Nyomtatás	Egy nyomtatás gomb segítségével kilistázzuk az írókat és könyveiket.
• Keresés:	Egy szerkesztődobozba begépeljük a keresett nevet, majd egy keresés gomb hatására indítjuk a keresést az *Írók* táblában.
Az írók karbantartását, nyomtatását és keresését egyetlen űrlapon fogjuk megvalósítani. A karbantartást és a keresést itt is egy *PageControl* különböző oldalaira helyezzük, a nyomtatást pedig egy gombbal oldjuk meg.	
Kiadók:	
• Karbantartás	Egy navigátorsor segítségével lépegetünk és szerkesztjük a kiadókat. Ezeket egy rácsban jelenítsük meg.
• Nyomtatás	A nyomtatás gomb segítségével kilistázzuk a kiadókat.
Témák:	
• Karbantartás	Egy navigátorsor segítségével lépegetünk és szerkesztjük a témaköröket. Ezeket egy rácsban jelenítsük meg.
• Nyomtatás	A nyomtatás gomb segítségével kilistázzuk a témaköröket.

Látható tehát, hogy alkalmazásunkban a négy fő funkciót négy űrlappal oldjuk meg. Az űrlapok mélyebb elemzése előtt azonban dolgozzuk ki alkalmazásunk menüszerkezetét a táblázatban felvázolt funkciók alapján. Természetesen még egy *Kilépés* menüpontot hozzá kell adnunk (lásd 10.5. ábra felső része).

A menünek egy másik lehetséges változatában a főmenüpontokat az alkalmazás funkciói alkotnák: *Karbantartás, Keresés, Nyomtatás, Kilépés* (lásd 10.5. ábra alsó része). Ez esetben az almenüpontokban felsorolnánk a funkciók tárgyát.

Mindkét változat jó, azt kell választanunk, amelyik a felhasználónak szimpatikusabb.

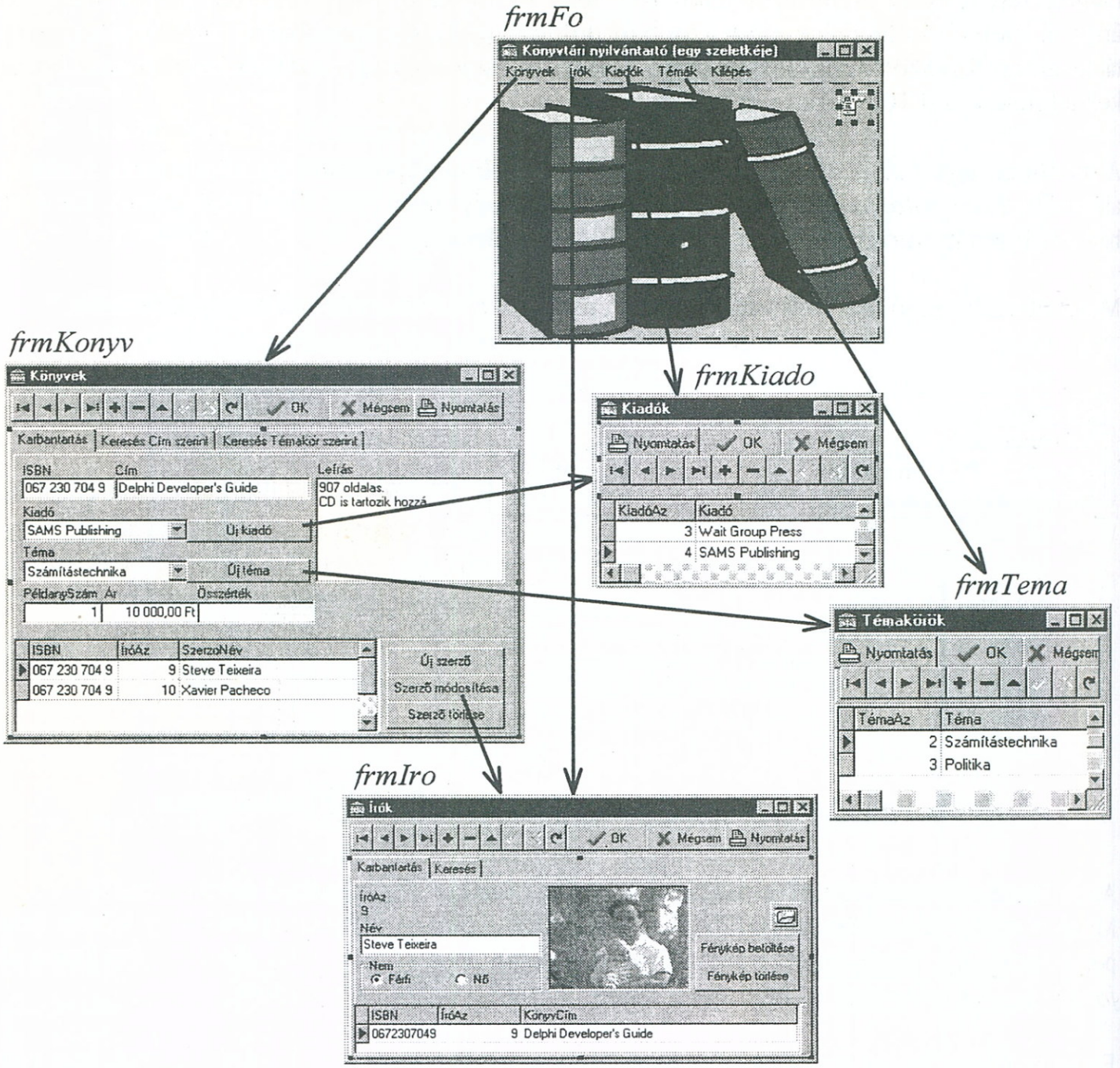
Könyvek		Írók	Kiadók	Témák	Kilépés
Karbantartás		Karbantartás	Karbantartás	Karbantartás	
Keresés ▶	Cím szerint	Keresés Név szerint	Nyomtatás	Nyomtatás	
Nyomtatás	Témakör szerint	Nyomtatás			

Karbantartás	Keresés		Nyomtatás	Kilépés
Könyvek	Könyvek ▶	Cím szerint	Könyvek	
Írók	Írók Név szerint	Témakör szerint	Írók	
Kiadók			Kiadók	
Témák			Témák	

10.5. ábra. A menüszerkezet két változata (Mi a felsőt választottuk.)

Az itt felvázolt menüt az alkalmazás főűrlapján (***frmFo***) fogjuk majd elhelyezni.
Minden egyes főmenüpont (a *Kilépést* leszámítva) összes alfunkciójának egy közös űrlapot készítünk. Tehát négy adatbázisos űrlapunk lesz: *frmKonyv*, *frmIro*, *frmTema* és *frmKiado*. Az űrlapokat és bejárási sorrendjüket a 10.6. ábra szemlélteti.

Elemezzük együtt az ábrát! A főűrlapról a megfelelő menüpontok segítségével, megjeleníthetjük a könyvek, írók, kiadók és témák űrlapokat. Alkalmazásunk lényege a könyvek nyilvántartása. Valószínűleg ez lesz a leghasználtabb űrlap. Amikor felvezetjük egy könyv adatait, akkor megadjuk a kiadóját, témáját és szerzőit is. Ha egy eddig még nem létező kiadóra vagy témára szeretnénk hivatkozni, akkor egy gomb segítségével lehessen átlépni a megfelelő űrlapra, hogy ott felvihessük az új adatokat. Tegyük fel, hogy felvettünk egy új kiadót a kiadók ablakban. Ha erről az *OK* gombbal lépünk vissza a könyvekhez, akkor az új kiadó íródjon is rögtön be az aktuális könyvhöz. Ugyanígy lehessen új témát és szerzőket is felvinni. Ezért van az, hogy a könyvek űrlapról mindhárom másik űrlapot el lehet érni. A könyvek űrlapot a legvégére fogjuk hagyni, hogy létrehozásakor a többi már működőképes legyen.



10.6. ábra. Űrlapjaink, ahogyan ezeket „megálmodtuk”
(a nyilak a bejárási sorrendet mutatják)

Keressük meg űrlapjaink közös elemeit, építsünk fel egy **űrlaphierarchiát** (10.7. ábra). Figyeljük meg a funkciók táblázatában az aláhúzásokat!

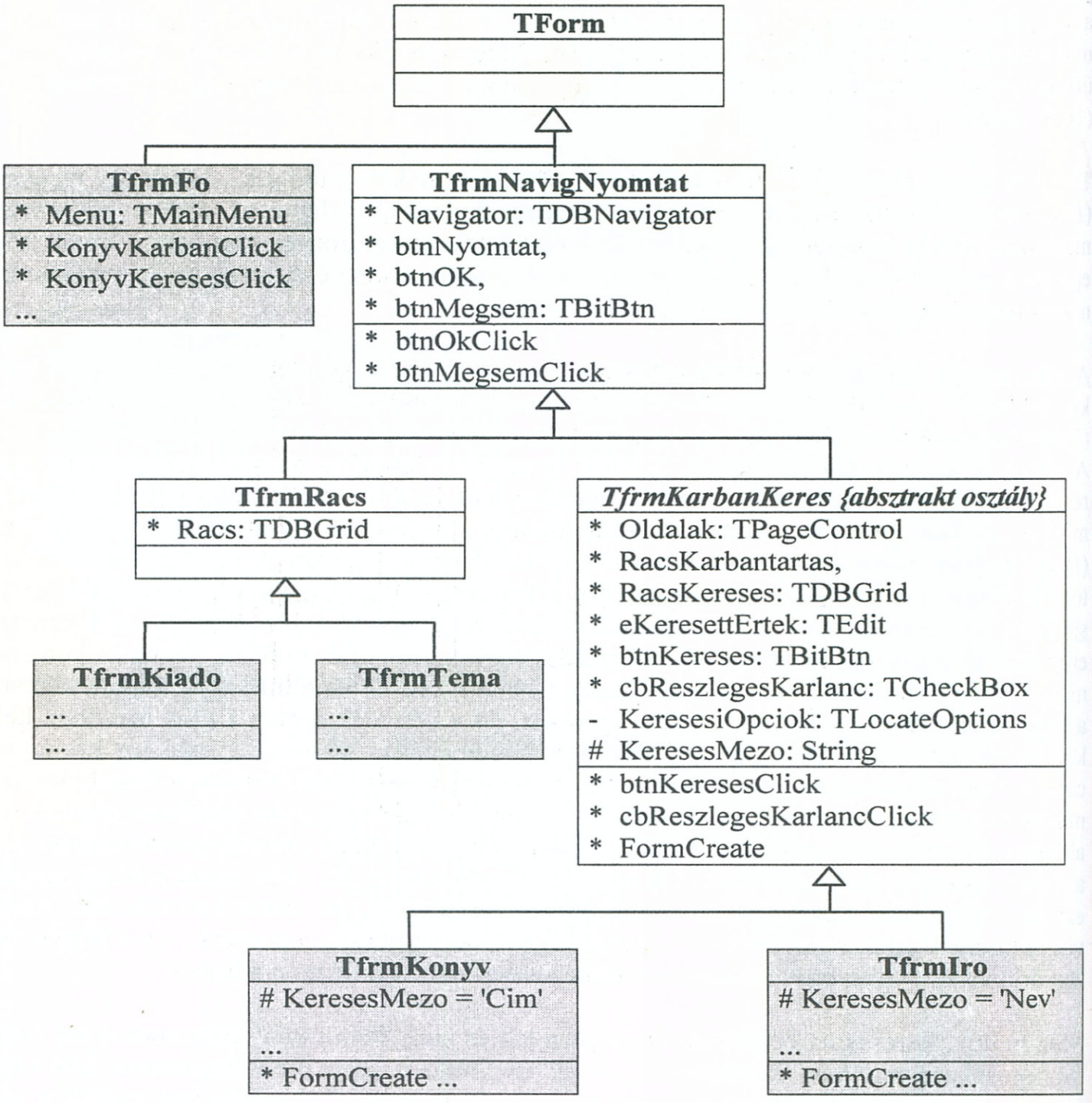
Azt vesszük észre, hogy navigátorsorra és nyomtatás gombra mind a négy űrlapon szükség van. Készítsünk tehát egy űrlapmintát, melyen egy navigátorsor, egy *Nyomtatás*, egy *Ok* és egy *Mégsem* gomb szerepel. Az *Ok* és *Mégsem* gombok segítségével tudjuk majd az űrla-

pokat bezárni. Az *Ok* hatására a még esetleg szerkesztés alatt álló rekordot lementjük, míg a *Mégsem*-re mentés nélkül zárjuk be az ablakot. E két gomb kattintására épített metódust már ebben az űrlaposztályban meg tudjuk írni. Megtervezzük tehát az ***TfrmNavigNyomtat*** űrlapot, majd lementjük mintaként. Minden további űrlapunkat ebből fogjuk származtatni (10.7. ábra).

További közös elemeket is felfedezhetünk. Mind a témákat, mind a kiadókat egy rácsban (*DBGrid*) jelenítjük meg. Emiatt hozzunk létre egy másik űrlapot a *TfrmNavigNyomtat* mintájára. Az új űrlapon (***TfrmRacs***) az örökölt navigátorsoron és gombokon kívül még egy rács is található. A *TfrmRacs* űrlapot is elmentjük mintaként, hogy majd ebből származtathassuk a *Téma* és *Kiadó* űrlapokat.

A könyvek és írók karbantartása és keresése egy *PageControl* külön oldalain történik. A két oldallal rendelkező *PageControl* komponenst érdemes egy közös „űrlap-ősbe" (***TfrmKarbanKeres***) kiemelni. És nemcsak ezt. Elemezzük az oldalakat!
A könyvek és írók karbantartását egy-egy fő-segéd űrlapon valósítjuk meg. Ennek felső része nem közös, hiszen a könyveknél egészen más és több mezőt kell megjelenítenünk, mint az íróknál. Az alsó részben viszont mind a könyveknél, mind az íróknál egy rács (*RacsKarbantartas*) lesz látható. Ezt a közös ősbe fogjuk helyezni. A keresés is hasonlóan történik: egy szerkesztődobozba begépeljük a keresett szöveget, majd egy gombbal rákeresünk. Itt nemcsak a szerkesztődobozt (*eKeresettErtek*), a gombot (*btnKereses*) és a keresés eredményét megjelenítő rácsot (*RacsKereses*) fogjuk kiemelni az ősbe, hanem a keresést megvalósító metódust is (*btnKeresesClick*). Ebben a *Locate* metódussal rá fogunk keresni a szerkesztődobozba begépelt értékre. Igen ám, de a könyvek esetén a *Cim* mezőben kell keresnünk, az íróknál pedig a *Nev* mezőben. Ez tehát nem közös. Ugyanakkor kár lenne csak emiatt elhalasztani a *btnKeresesClick* metódus megírását, hiszen a későbbiekben majdnem ugyanazt kellene begépelnünk két példányban (vagy ha továbbfejlesztjük az alkalmazást, még több példányban is). Vezessünk tehát itt be egy védett (*protected*) mezőt, a *KeresesMezo*-t. A *btnKeresesClick*-ben erre hivatkozunk. Az utód osztályokban csupán ezt az értéket kell majd beállítanunk a megfelelő mezőre, és máris kereshetünk. Amíg viszont nem állítjuk be semmire a *KeresesMezo*-t, addig nem kereshetünk, így a megtervezett űrlaposztály absztrakt lesz, csak örökítésen keresztül „állja meg a helyét".

Mint tudjuk, keresésnél beállíthatjuk, hogy a kis és nagy betűk különbözzenek-e és, hogy részleges karakterláncra keressünk-e. A mi esetünkben eleve ne számítsanak különbözőnek a kis és nagy betűk (ezt az *frmKarbanKeres* űrlap *OnCreate*-ben beégetjük), a másik opciót pedig a felhasználóra bízzuk: helyezzünk az űrlapra egy jelölőnégyzetet (*cbReszlegesKarlanc*). A jelölőnégyzet kattintására (*cbReszlegesKarlancClick*) beállítjuk egy privát mezőbe (*KeresesiOpciok*-ba) az éppen kiválasztott keresési opciót. Kereséskor (*btnKeresesClick*) pedig ehhez tartjuk magunkat.
Így készül el a ***TfrmKarbanKeres***. Ebből fogjuk a Könyv és Író űrlapokat származtatni.



10.7. ábra. Az űrlapok osztályhierarchiája.
Csak a besatírozott osztályokat példányosítjuk.

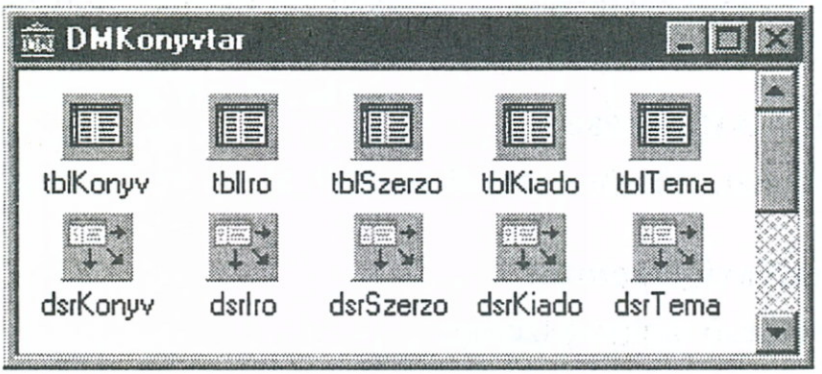
10.5 Az alkalmazás kivitelezése

Immár megterveztük és létrehoztuk az adatbázist, kigondoltuk alkalmazásunk menüszerkezetét és űrlaphierarchiáját. Nekiláthatunk a kivitelezésnek! Kezdjünk egy új alkalmazást. Első lépésben felépítjük ennek adatmodulját, majd sorban létrehozzuk az űrlapokat is.

10.5.1 Az adatmodul felépítése

Lépések:

- Hozzunk létre egy adatmodult. Nevezzük el *DMKonyvtar*-nak.
- Helyezzünk el minden egyes tábla számára egy-egy *TTable* és *TDataSource* komponenst. Nevezzük el ezeket a 10.8. ábra szerint.



10.8. ábra. Az adatmodul (kezdetleges állapotában)

- Hozzuk létre a perzisztens mezőket. Minden egyes táblakomponens esetén hívjuk elő a mezőszerkesztőt, és töltsük be a fizikai táblában létező mezőket. A beolvasott mezők tulajdonságait már tervezési időben testre szabhatjuk:
  - Minden mező *DisplayLabel* jellemzőjét állítsuk a mező ékezetes nevére.
  - Az *ISBN* mezőnél (ez két táblában is megtalálható) állítsuk be az *EditMask* jellemzőjét '000\ 00\ 0000\ 0;0;_'–ra. Ezzel azt mondjuk meg, hogy az ISBN szám pontosan 10 számjegyből áll, a maszk szerinti csoportosításban. A három bevezetett szóköz miatt növelnünk kell a *DisplayWidth* értékét 13-ra.
  - A *tblIro Nem* mezőjének állítsuk be *DisplayValues* jellemzőjét *'Férfi;Nő'* -re.
  - A *tblKonyv*-ben hozzunk létre egy számított mezőt, az *Osszertek*-et. Az új mező kiszámolásának algoritmusát a *tblKonyv* táblakomponens *OnCalcFields* eseményébe írjuk:

```
procedure TDMKonyvtar.tblKonyvCalcFields(DataSet: TDataSet);
begin
  With DMKonyvtar Do
    tblKonyvOsszertek.Value:=tblKonyvAr.Value*
                tblKonyvPeldanyszam.Value;
end;
```

- A *tblKonyv.IndexFieldNames* jellemzőjét állítsuk be *'Cim'*-re, hiszen a könyveket a címük szerinti rendezettségben szeretnénk látni.
- Az írók névsorrendben jelenjenek meg, tehát *tblIro.IndexFieldNames := 'Nev'*.
- Nyissuk meg a táblákat (*Active := True*). Majd a fejlesztés legvégén be fogjuk ezeket zárni. Ekkor a megnyitásukat az adatmodul *OnCreate*-jébe, a bezárásukat pedig az *OnDestroy*-ba építjük majd be. De addig is „jól esik" már tervezési időben látni az adatokat.
- Állítsuk az adatforrások *AutoEdit* jelemzőjét *False*-ra. Így a felhasználó, ha szerkeszteni kívánja az adatokat, csak egy adott gomb lenyomása után teheti ezt. Elkerüljük ezáltal a véletlen módosításokat.

Nyilván lesznek még további beállítások, származtatott mezők... Egyelőre ennyi a biztos.

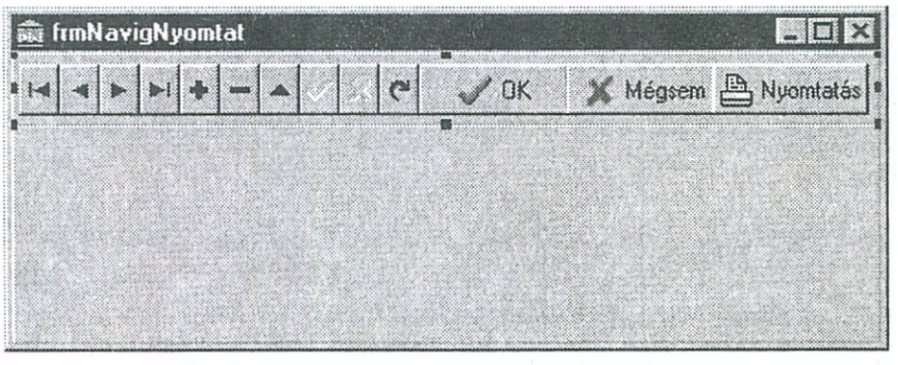
10.5.2 Az űrlapok kivitelezése

Vegyük sorba az űrlapokat, először a mintákat (10.7. ábra).

10.5.2.1 TfrmNavigNyomtat

Tervezzük meg a következő táblázat szerint:

Komponens	Jellemző	Érték
pnGombok: TPanel	Align	alTop
pnEgyebek: TPanel	Align	alClient
Navigator: TDBNavigator	DataSource	még nem tudjuk
btnOK: TBitBtn	Kind	bkOK
btnMegsem: TBitBtn	Kind	bkCancel
	Caption	'Mégsem'
btnNyomtatas: TBitBtn	Caption	'Nyomtatás'
	Glyph	DELPHI\IMAGES\BUTTONS\PRINT.BMP



10.9. ábra. A *TfrmNavigNyomtat* űrlapminta

Az *OK* és *Mégsem* gombokat már le is kódolhatjuk. A mentés vagy visszavonás abban a táblában történik, amelyre a navigátorsor a *DataSource* jellemzőjén keresztül mutat.

```
procedure TfrmNavigNyomtat.btnOkClick(Sender: TObject);
begin
  {Ha szerkesztésben voltunk, akkor most mentjük a módosításokat, majd
  bezárjuk az űrlapot (ez automatikus a gomb ModalResult-ja miatt).}
  With Navigator.DataSource.DataSet Do
    If State in [dsInsert, dsEdit] Then
      Post;
end;

procedure TfrmNavigNyomtat.btnMegsemClick(Sender: TObject);
begin
  {Ha szerkesztésben voltunk, akkor most visszavonjuk a módosításokat,
   majd utána bezárjuk az űrlapot }
  With Navigator.DataSource.DataSet Do
    If State in [dsInsert, dsEdit] Then
      Cancel;
end;
```

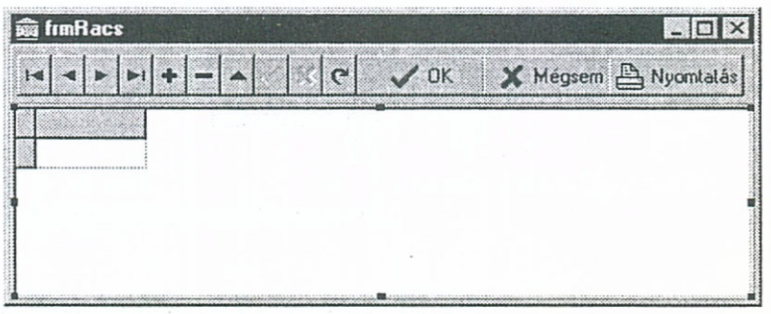
A *dsEdit*, *dsInsert* konstansok definíciói a DB egységben találhatók. Építsük be ezt az egységet a használt egységek közé. (*Uses DB;*)

Az így elkészített űrlapot most mentsük le az űrlapminták közé a gyorsmenü *Add To Repository* parancsával.

10.5.2.2 TfrmRacs

A *TfrmNavigNyomtat*-ból származtatjuk. Hívjuk meg a *File/New...* menüpontot, azon belül pedig a *Forms* lapot. Válasszuk ki az űrlapmintát, és jelöljük be az *Inherit* választógombot.

Az így létrehozott űrlap ősének minden elemét örökli, sőt, ha később az ősben változtatnánk valamit, akkor ez az utódjaira is kiterjedne.
Az új űrlapra csak egy rácsot (*DBGrid*) kell elhelyeznünk a *pnEgyebek* panelre. Az igazítását állítsuk be *alClient*-ra.



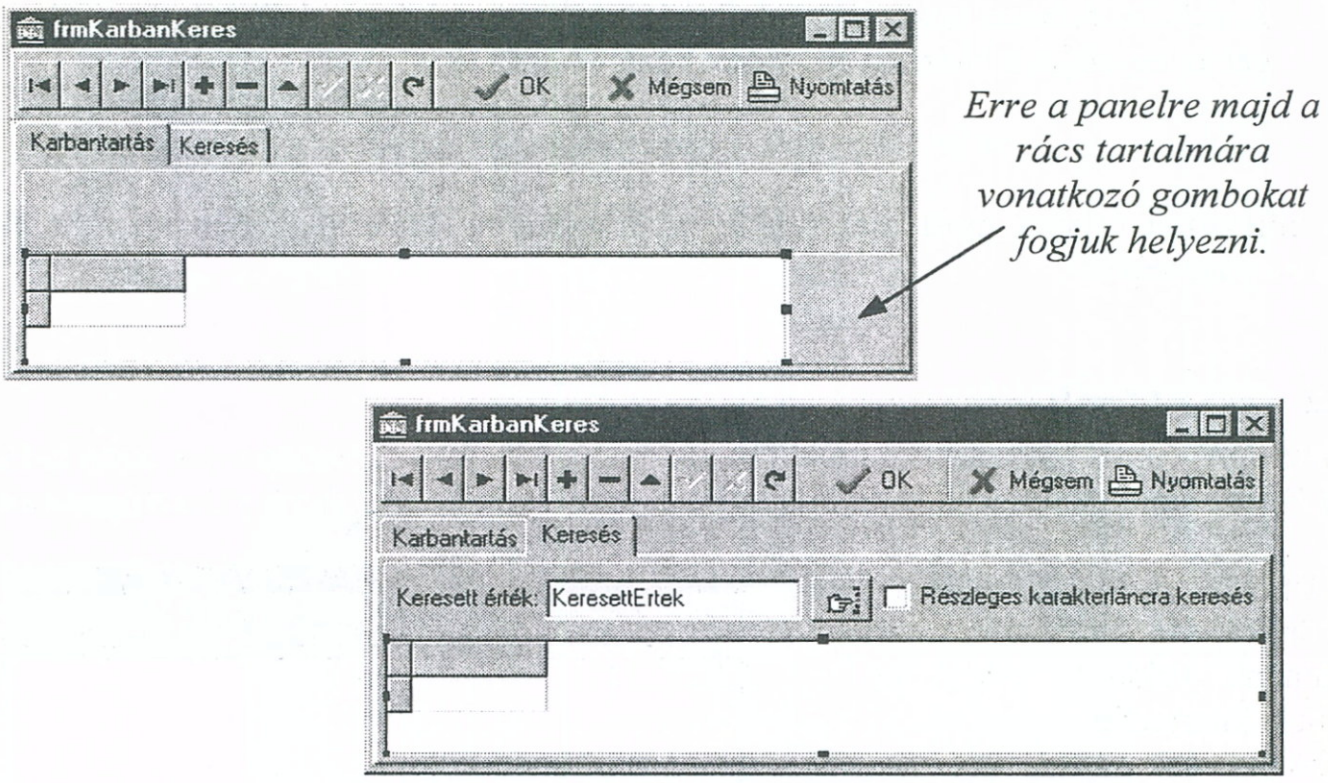
10.10. ábra. A *TfrmRacs* űrlapminta

Mentsük el az *frmRacs* űrlapunkat is a minták közé.

10.5.2.3 TfrmKarbanKeres

Ezt is a *TfrmNavigNyomtat* mintájára tervezzük meg. Az új elemeket a következő táblázat mutatja:

Komponens	Jellemző	Érték
Oldalak: TPageControl	Align	alClient
A Karbantartás oldalra:		
pnFoUrlap:TPanel	Align	alTop
pnSegedUrlapGombok:TPanel	Align	alRight
RacsKarbantartas:TDBGrid	Align	alClient
	ReadOnly	True
A Keresés oldalra:		
pnKereses:TPanel	Align	alTop
eKeresettErtek:TEdit		
btnKereses:TBitBtn	Caption	''
	Glyph	DELPHI\IMAGES\ BUTTONS\FIND.BMP
cbReszlegesKarlanc:TCheckBox		
RacsKereses:TDBGrid	Align	alClient



10.11. ábra A *TfrmKarbanKeres* űrlap mindkét „arca”

Amint a 10.11. ábra is mutatja, a *RacsKarbantartas* mellett fenntartottunk helyet az esetleges gomboknak, melyekkel a rács tartalmát átírhatjuk. A könyvek űrlapon ide fogjuk

helyezni az *Új szerző*, *Szerző módosítása* és *Szerző törlése* gombokat (10.6. ábra, 242. oldal). Az írók űrlapon a rácsban az éppen aktuális író könyvei fognak megjelenni, és ezt az információt itt nem szerkeszthetjük. Emiatt majd a *pnSegedUrlapGombok* panelt el is fogjuk tüntetni.
Egyik űrlapon sem lesz tehát szükség a rács **közvetlen** szerkesztésére. Állítsuk máris be *ReadOnly* jellemzőjét Igazra.

Vegyük fel az űrlaposztályba az osztályhierarchián (10.7. ábra, 244. oldal) feltüntetett saját mezőket, majd kódoljuk le mindazt, amit csak lehet:

```
type
  TfrmKarbanKeres = class(TfrmNavigNyomtat)
    ...
    private
      KeresesiOpciok:TLocateOptions;
    protected
      KeresesMezo:String;
  end;

...
procedure TfrmKarbanKeres.FormCreate(Sender: TObject);
begin
  inherited; {ezt nem szabad kitörölni}
  KeresesiOpciok := [loCaseInsensitive];
end;

procedure TfrmKarbanKeres.ReszlegesKarlancClick(Sender: TObject);
begin
  inherited;
  If (Sender as TCheckbox).Checked Then
    KeresesiOpciok:= KeresesiOpciok + [loPartialKey]
  Else
    KeresesiOpciok:= KeresesiOpciok - [loPartialKey];
end;

procedure TfrmKarbanKeres.btnKeresesClick(Sender: TObject);
begin
  inherited;
  With Navigator.Datasource.Dataset Do
    If Not Locate (KeresesMezo, eKeresettErtek.text,KeresesiOpciok)
      Then ShowMessage('Nincs találat!');
end;
```

Mentsük el a kész űrlapot az űrlapminták közé.

Előfordulhat, hogy a fordító megakad a *TLocateOptions* típusnál. Ez a *DB* egységben van definiálva, írjuk ezt is be a *Uses* -ba.

Az űrlapmintákkal immár elkészültünk. Most következnek tényleges ablakaink (az űrlaphierarchia besatírozott osztályai, 10.7. ábra).

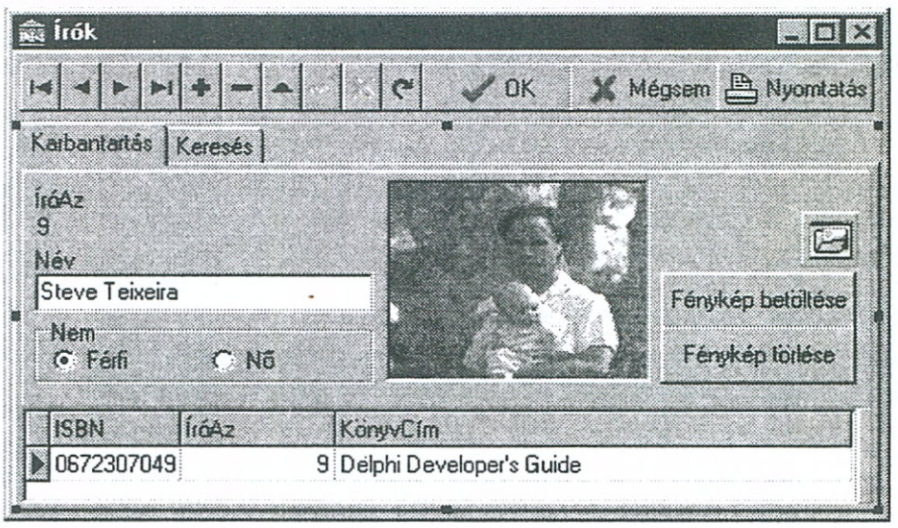
10.5.2.4 TfrmFo

A főűrlap a *TForm* osztály leszármazottja lesz. Tervezzük meg a menüjét a 10.5. ábra (241. oldal) felső menüszerkezete alapján. Az űrlap üresen tátongó részére betölthetünk egy képet.

10.12. ábra. Alkalmazásunk főűrlapja

10.5.2.5 TfrmIro

Hozzunk létre egy új űrlapot a *TfrmKarbanKeres* alapján. Nevezzük el *frmIro*-nak. Továbbfejlesztését kezdjük a *Karbantartás* oldallal. Ennek felső részében helyezzük el az író adatait megjelenítő elemeket.



10.13. ábra. Írók karbantartása

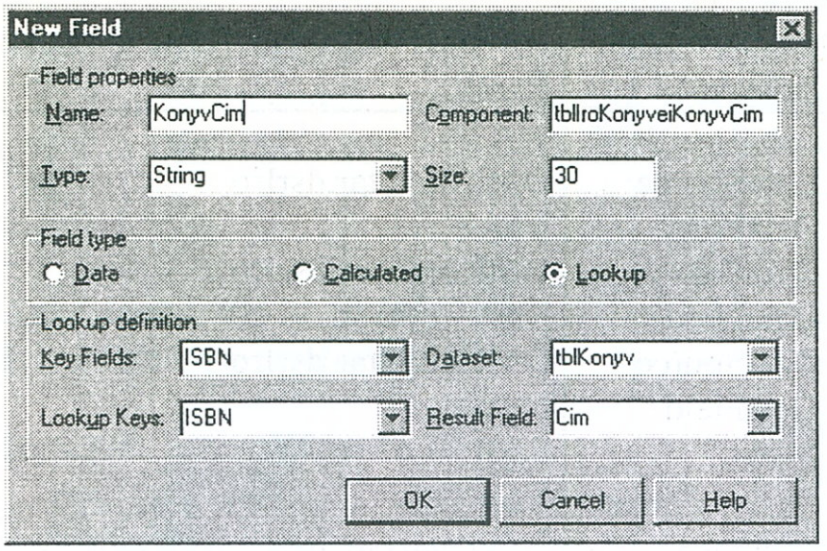
Komponens	Jellemző	Érték
A Karbantartás oldalra:		
dbtIroAz: TDBText	DataSource	DMKonyvtar.dsrIro
	DataField	IroAz
dbeNev: TDBEdit	DataSource	DMKonyvtar.dsrIro
	DataField	Nev
dbrgNem: TDBRadioGroup	DataSource	DMKonyvtar.dsrIro
	DataField	Nem
	Items	Férfi
		Nő
	Columns	2 *{egymás mellett – nem egymás alatt – jelenjenek meg}*
dbiFenykep: TDBImage	DataSource	DMKonyvtar.dsrIro
	DataField	Fenykep
btnFenykepBetoltese,	Caption	'Fénykép betöltése',
btnFenykepTorlese: TButton		'Fénykép törlése'
OpenPictureDialog:		*{A fénykép futáskori betöltéséhez*
TOpenPictureDialog		*szükséges}*

A rácsban az aktuális író könyveit szeretnénk látni. Ismerjük tehát az *IroAz*-t, meg kellene tudni, hogy milyen könyveket írt. A keresett információ a *Szerző* táblában található (*tblSzerzo*), tehát a rácsot (*RacsKarbantartas*) a *dsrSzerzo*-re kell irányítanunk.
Ha azt szeretnénk, hogy a rács csak az aktuális író könyveit mutassa, akkor a *tblSzerzo* tábla tartalmát meg kell szűrnünk a *tblIro*-ból származó *IroAz* mező értékei alapján. Lépjünk át az adatmodulra, jelöljük ki a *tblSzerzo* komponenst, és végezzük el a következő két beállítást (ne feledkezzünk meg az indexről):

```
IndexFieldNames:= 'IroAz;ISBN';
MasterSource:= dsrIro;
MasterField:= 'IroAz';
```

Ezek után, mivel a *tblSzerzo* táblakomponensünk már csak a *tblIro* aktuális írójának könyveit tartalmazza, azt javaslom, hogy a *tblSzerzo*-t nevezzük át *tblIroKonyvei*-re, a *dsrSzerzo*-t pedig *dsrIroKonyvei*-re. (Így később is első ránézésre tudni fogjuk, hogy ez egy író szerint szűrt szerző tábla.) Ne aggódjunk, az új nevet a rendszer automatikusan átvezeti a különböző űrlapelemek hivatkozásaiban. (A mi általunk írt kódban már nem tenné.)

A rácsban immár láthatók az író könyveinek ISBN számai. Tulajdonképpen a könyvek címeit is jó lenne látni. Hozzunk létre egy származtatott mezőt a *tblIroKonyvei* táblában. A *KonyvCim* egy „kikeresett" (*lookup*) mező lesz, hiszen a *tblIroKonyvei* -ben levő *'ISBN'* alapján kikereshető a *tblKonyv*-ből.
Az új mező beállításait a 10.14. ábra mutatja.




```
LookupDataset := tblKonyv;
LookupKeyFields:= 'ISBN';
KeyFields:= 'ISBN';
LookupResultField:= 'Cim';
```

Az újonnan létrehozott mező *DisplayLabel* tulajdonságát állítsuk *'KönyvCím'*-re, ez fog megjelenni a rács fejlécében.

10.14. ábra. A könyv címének kikeresése

A *RacsKarbantartas* melletti *pnSegedUrlapGombok* panelre most nincs szükség, tüntessük is ezt el (*Visible := False*). Panelünk tervezéskor ugyan még nem, de majd futási időben láthatatlan lesz.

Kódoljuk le a fénykép betöltését és kitörlését:

```
procedure TfrmIro.btnFenykepBetolteseClick(Sender: TObject);
begin
   inherited;
   If OpenPictureDialog.Execute Then
      With DMKonyvtar, tblIro Do
         begin
            If Not (State In [dsEdit, dsInsert]) Then
               Edit;
            tblIroFenykep.LoadFromFile(OpenPictureDialog.FileName);
            Post;
         end
end;

procedure TfrmIro.btnFenykepTorleseClick(Sender: TObject);
begin
   inherited;
   With DMKonyvtar, tblIro Do
      begin
         If Not (State In [dsEdit, dsInsert]) Then
            Edit;
         tblIroFenykep.Clear;
         Post;
      end
end;
```

Ezzel a karbantartás oldal készen is van, következzen a keresés: állítsuk be a *RacsKereses* adatforrását a *DMKonyvtar.tblIro*-ra. A keresés kódját örököljük az ős űrlaposztályból (Hurrá!). Nekünk most csak a *KeresesMezo* értékét kell beállítanunk a *Nev*-re.

```
procedure TfrmIro.FormCreate(Sender: TObject);
begin
  inherited;
  KeresesMezo:= 'Nev';
end;
```

Az *frmIro* űrlap elkészült. Ahhoz, hogy kipróbálhassuk, lépjünk át a főűrlapra és kódoljuk le ennek írókat érintő menüpontjait.
A *'Karbantartás'* és *'Keresés Név szerint'* menüpontok az *frmIro* űrlap különböző oldalait jelenítik meg. Mindkét menüpont ugyanazt a metódust hívja meg *(IroClick)*, abban pedig a *Tag* jellemzőjüknek megfelelő oldalt tesszük aktívvá.

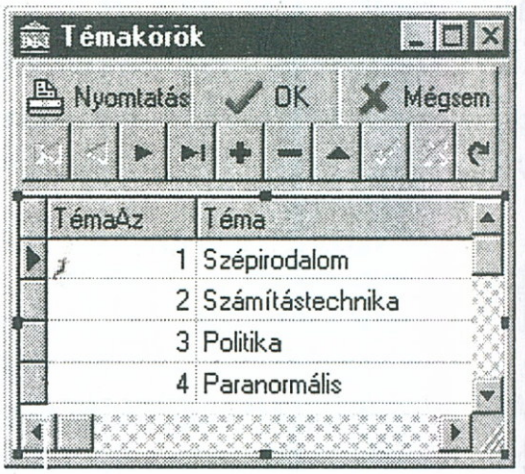
Komponens (Írók/~ menüpont)	Jellemző	Érték
Karbantartás	Tag	0
	OnClick	IroClick
Keresés Név szerint	Tag	1
	OnClick	IroClick
Nyomtatás	OnClick	IroNyomtatas

```
procedure TfrmFo.IroClick(Sender: TObject);
begin
  With frmIro Do
    begin
      Oldalak.ActivePage:= Oldalak.Pages[(Sender as TMenuItem).Tag];
      ShowModal;
    end;
end;

{Majd amikor a nyomtatást is implementáljuk, elég lesz csak a
btnNyomtatas.OnClick-jét megírni}
procedure TfrmFo.IroNyomtatas(Sender: TObject);
begin
  frmIro.btnNyomtatasClick(Sender);
end;
```

Futassa le az alkalmazást. Próbáljon új írókat felvinni, létezők adatait módosítani, törölni. Ha egy írónak már könyve is van, akkor őt nem törölhetjük adatbázisunkból.

10.5.2.6 TfrmTema

A témák karbantartási űrlapját a *TfrmRacs* űrlapmintából származtatjuk.
Elég a *Navigator* és a *Racs* komponensek *DataSource* jellemzőjét a *DMKonyvtar.dsrTema*-ra állítanunk, és az űrlap máris működőképes. Esetleg annyit tehetünk még, hogy mivel a rácsban alig két oszlop található, a navigátorsort és a gombokat kicsit átrendezzük; így az űrlap keskenyebb lesz.



10.15. ábra. *TfrmTema* űrlapunk

A főűrlap menüpontjait az alábbiak szerint kódoljuk le:

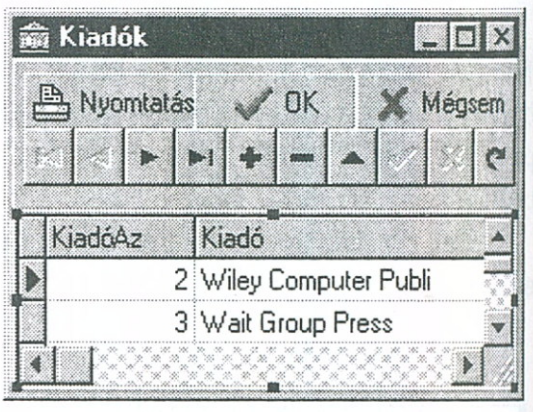
```
procedure TfrmFo.TemaKarbantartas(Sender: TObject);
begin
   frmTema.ShowModal;
end;
procedure TfrmFo.TemaNyomtatas(Sender: TObject );
begin
   frmTema.btnNyomtatasClick(Sender);
end;
```

Próbáljuk ki a téma űrlapot! Vigyünk fel egy új témakört, töröljünk, és módosítsunk.

10.5.2.7 TfrmKiado

Az *frmTema* űrlap elkészítésével azonos módon tervezzük meg a kiadó űrlapot is (*frmKiado*).

Ne felejtkezzünk el ennek főűrlapról történő megjelenítéséről sem!

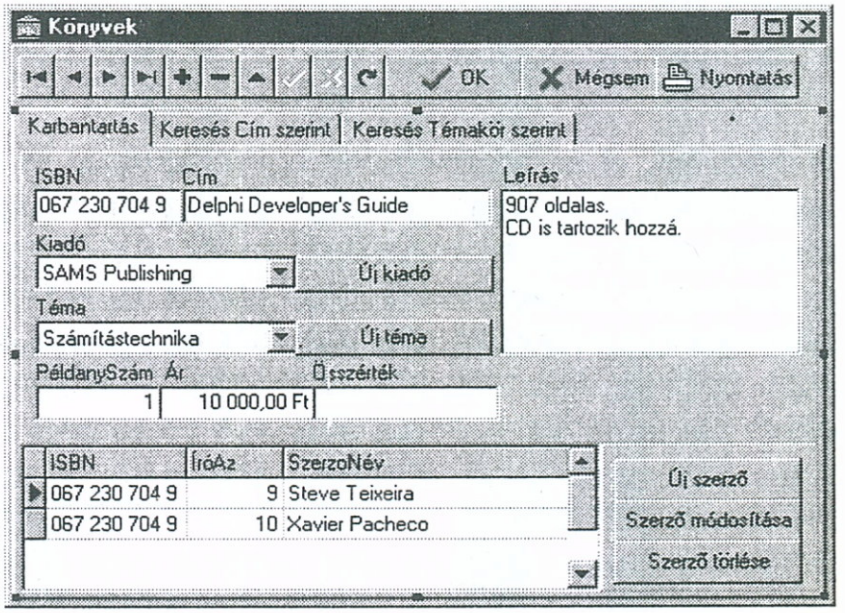


10.16. ábra. *TfrmKiado* űrlapunk

10.5.2.8 TfrmKonyv

Könyvek űrlapunkat a *TfrmKarbanKeres* űrlapmintából származtatjuk.

Tervezzük meg előbb a *Karbantartás* oldalát. Jelenítsük meg a *tblKonyv* tábla mezőszerkesztőjét, majd a mezőket vonszoljuk egyenként az űrlapra. Minden mező számára a rendszer automatikusan létrehoz egy címkét (*Label*) és egy szerkesztődobozt *(DBEdit)*, amiben azonnal látható az illető mező értéke. A kiadót és témát kombinált listából szeretnénk majd kiválogatni, így ezekhez *DBLookupComboBox* komponenseket kell használnunk.



10.17. ábra. Könyvek karbantartása egy fő-segéd űrlapon

Beállításukat a következő táblázat mutatja:

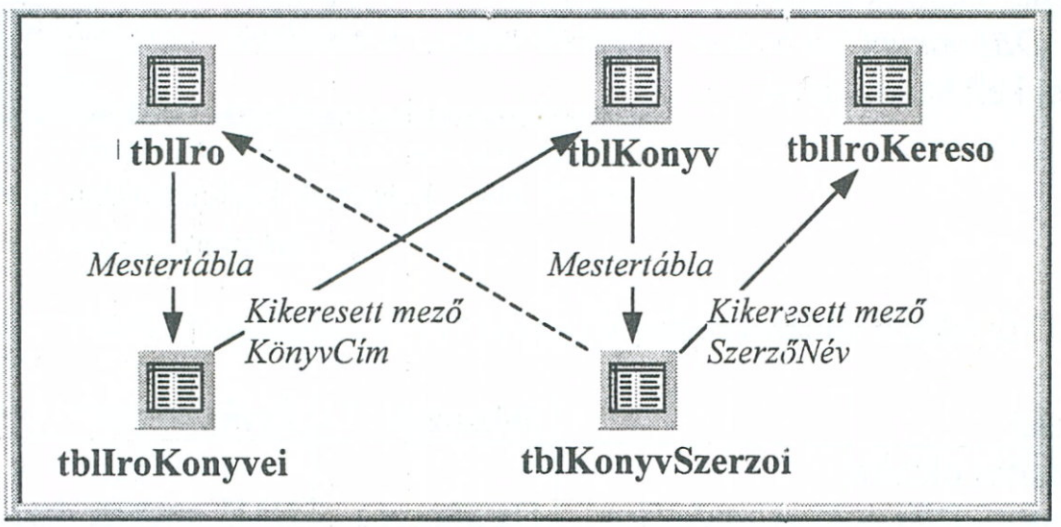
Komponens	Jellemző	Érték
A Karbantartás oldalra:		
dblcKiado: TDBLookupComboBox	DataSource	DMKonyvtar.dsrKonyv
	DataField	KiadoAz
	ListSource	DMKonyvtar.dsrKiado
	ListField	Kiado
	KeyField	KiadoAz
dblcTema: TDBLookupComboBox	DataSource	DMKonyvtar.dsrKonyv
	DataField	TemaAz
	ListSource	DMKonyvtar.dsrTema
	ListField	Tema
	KeyField	TemaAz
dbmLeiras: TDBMemo	DataSource	DMKonyvtar.dsrKonyv
	DataField	Leiras
btnUjKiado: TButton	Caption	'Új kiadó'
btnUjTema: TButton	Caption	'Új téma'
btnUjSzerzo: TButton	Caption	'Új szerző'
btnSzerzoModositasa: TButton	Caption	'Szerző módosítása'
btnSzerzoTorlese: TButton	Caption	'Szerző törlése'

A rácsban csak az aktuális könyv szerzőit szeretnénk látni. Ezt az információt a SZERZO.DB-ből olvashatjuk ki. Van már adatmodulunkon egy erre irányított táblakom-

ponens (*tblIroKonyvei*), de ez itt nem használható. Hozzunk létre egy új táblakomponenst (*tblKonyvSzerzoi*), irányítsuk a SZERZO.DB-re, majd szűrjük a *tblKonyv* tábla *ISBN* mezője szerint.

```
tblKonyvSzerzoi.MasterSource := dsrKonyv;
tblKonyvSzerzoi.MasterField := 'ISBN';
```

Helyezzünk el az adatmodulon egy adatforrást is (*dsrKonyvSzerzoi*); a rács ebből fogja adatait kiolvasni. Annak érdekében, hogy a rácsban még a szerző neve is megjelenjen, hozzunk létre egy származtatott mezőt a *tblKonyvSzerzoi* táblában. A *'SzerzoNev'* egy „kikeresett" (*lookup*) mező lesz, hiszen a *tblKonyvszerzoi*-ben levő *'SzerzoAz'* alapján kikereshető az IRO.DB-ből. Van már az adatmodulon egy erre irányuló táblakomponensünk (*tblIro*), de ennek használata **egy veszélyes hibalehetőséget** rejt magában (10.18. ábra).



10.18. ábra. Körkörös hivatkozás. A nyilak azt mutatják, hogy egy tábla milyen más táblára van kihatással.

A *tblIroKonyvei* a mestertábla kapcsolat miatt mindig csak a *tblIro*-ban levő aktuális író könyveit tartalmazza. A *tblIroKonyvei* táblában a könyvek címeit a *tblKonyv*-ból kerestetjük ki. Konkrét feladatunkban nem lehet a *tblIroKonyvei*-t szerkeszteni, de egy pillanatra feltételezzük, hogy ez nem így van. Ha tehát a *tblIroKonyvei* tábla egy rekordjában beállítunk egy másik könyvet (például úgy, hogy a rács *KönyvCím* oszlopának lebomló listájából kiválasztunk egy másik címet), akkor a *tblKonyv*-ben ez lesz az aktuális könyv. Ez azonnal maga után vonja a *tblKonyvSzerzoi* tábla tartalmának váltását is. Ha ekkor, az író nevének kikereséséért a *tblIro*-hoz fordulnánk, akkor visszajutottunk oda, ahonnan elindultunk, bezárul a kör.
Az ilyen rejtett körkörös hivatkozásokat a Delphi nem veszi észre (egyébként is elég nehéz lenne). Ezekre nekünk kell nagyon vigyáznunk.

A „kikeresett" mezők keresőtáblájaként csak akkor használjunk egy már létező (máshol is használt, megjelenített) táblakomponenst, ha leellenőriztük, hogy nem vezet körkörös hivatkozáshoz.

Hatékonysági meggondolások

Általában jó, ha a „kikeresett" (*lookup*) mezők értékeit külön táblakomponensekből nézzük ki. Ezekhez *DataSource* nem is fog tartozni, hiszen tartalmát közvetlenül nem jelenítjük meg. Azonban azt is szem előtt kell tartanunk, hogy minél több táblakomponens van alkalmazásunkban, annál jobban lelassul az egész. A nagyobb alkalmazásoknál már ajánlatos elgondolkodnunk azon, hogy hol indokolt a külön keresőtábla felvétele. Sőt! Nem minden táblát kell az alkalmazás futása alatt végig nyitva tartanunk. Nagyon nehéz általános szabályt kialakítani, hiszen az sem jó, ha minden táblát az űrlap megjelenítésekor megnyitunk, elrejtésekor pedig bezárunk, mivel a nyitás-zárás is időigényes. Okosan kell döntenünk arról, hogy az alkalmazásban mely űrlapok jöjjenek létre automatikusan és melyek nem, valamint arról is, hogy mely táblákat nyitjuk meg állandó jelleggel, és melyeket csak ideiglenesen.

A mi esetünkben tehát biztonságosabb, ha létrehozunk adatmodulunkon egy újabb táblakomponenst, a *tblIroKereso*-t, és ezt az IRO.DB állományra irányítjuk. Ezt használjuk a *tblKonyvSzerzoi* táblában az *'SzerzoNev'* mező keresőtáblájaként.
Az új mező adatai:

```
LookupDataset := tblIroKereso;
LookupKeyFields:= 'IroAz';
KeyFields:= 'IroAz';
LookupResultField:= 'Nev';
```

A frissen létrehozott mező *DisplayLabel* tulajdonságát állítsuk 'Név'-re, ez fog megjelenni a rács fejlécében.

Egy **könyv kiadójának beállítása**kor két dolgot tehetünk:

- Ha egy már nyilvántartásba vett kiadót akarunk beállítani, akkor ezt a *dblcKiado* listájából válasszuk ki.
- Ellenkező esetben, ha a kiadó még nem szerepel a nyilvántartásunkban, akkor az *Új kiadó* gomb segítségével jelenítsük meg az *frmKiado* űrlapot. Ott vigyük fel az új kiadó adatait, majd az *OK* gombra kattintva térjünk vissza a könyvekhez. Ha a *Mégsem* gombbal zárnánk be a kiadó űrlapot, akkor ez azt jelentené, hogy meggondoltuk magunkat, már nem akarjuk átállítani a könyv kiadóját.

Kódoljuk mindezt le:

```
procedure TfrmKonyv.btnUjKiadoClick(Sender: TObject);
begin
   inherited;
   DMKonyvtar.tblKiado.Append; {új, üres rekord az új kiadó számára}
   If frmKiado.ShowModal = mrOk Then
   {ha OK-val léptünk vissza, akkor átírjuk az aktuális könyv kiadóját}
      begin
         If Not (DMKonyvtar.tblKonyv.State in [dsEdit, dsInsert]) Then
            DMKonyvtar.tblKonyv.Edit;
         DMKonyvtar.tblKonyvKiadoAz.Value:=
                              DMKonyvtar.tblKiadoKiadoAz.Value;
      end;
end;

{hasonló módon járunk el az új témakör felvitelénél is}
procedure TfrmKonyv.btnUjTemaClick(Sender: TObject);
begin
   inherited;
   DMKonyvtar.tblTema.Append;
   If frmTema.ShowModal = mrOk Then
      begin
         If Not (DMKonyvtar.tblKonyv.State in [dsEdit, dsInsert]) Then
            DMKonyvtar.tblKonyv.Edit;
            DMKonyvtar.tblKonyvTemaAz.Value:=
                              DMKonyvtar.tblTemaTemaAz.Value;
      end;
end;
```

Most nézzük, hogyan lehet **egy könyv szerzőit felvenni, módosítani, törölni.** Ha egy új szerzőt akarunk felvenni (például egy könyvnek most visszük fel a második szerzőjét), akkor meg kell jelenítenünk az *frmIro* űrlapot a *Keresés* oldalával, hogy ott a felhasználó kereshesse ki a megfelelő írót. Ha ez még nem szerepelne a nyilvántartásunkban, akkor átléphet a *Karbantartás* oldalra, és ott felviheti az új író adatait. Ezek után, ha *OK*-val lép ki az *frmIro*-ból, akkor a rácsba be kell szúrnunk egy új rekordot, mellyel jelezzük, hogy az aktuális könyvnek még a frissen beállított író is szerzője.
Ugyanez történik a 'Szerző módosítása' gomb hatására is, azzal a különbséggel, hogy itt nem viszünk fel egy új rekordot sem a szerző táblában, hanem az aktuálisban átírjuk az író azonosítóját.
A 'Szerző törlése' gomb hatására egyszerűen ki kell törölnünk a *tblKonyvSzerzoi* tábla aktuális rekordját.

```
procedure TfrmKonyv.btnUjSzerzoClick(Sender: TObject);
begin
   inherited;
   With DMKonyvtar, frmIro Do
      begin
         {Aktivvá tesszük az frmIro keresés lapját, majd megjelenítjük}
```

```
      Oldalak.ActivePage:= Oldalak.Pages[1];
      If frmIro.ShowModal=mrOk Then
         begin
            tblKonyvSzerzoi.Append;
            {a könyv ISBN száma a tblKonyv aktuális rekordjából
            származik, az IroAz pedig az éppen beállított írótól}
            tblKonyvSzerzoiISBN.Value:= tblKonyvISBN.Value;
            tblKonyvSzerzoiIroAz.Value:=tblIroIroAz.Value;
            tblKonyvSzerzoi.Post;
         end;
   end;
end;

procedure TfrmKonyv.btnSzerzoModositasaClick(Sender: TObject);
begin
  inherited;
  With frmIro, DMKonyvtar Do
    begin
      Oldalak.ActivePage:= Oldalak.Pages[1];
      IF ShowModal = mrOk Then
         begin
            {Ha OK-val lépett vissza, akkor átírjuk az író
            azonosítóját arra, amit ott kiválasztott}
            If Not (tblKonyvSzerzoi.State in [dsEdit, dsInsert]) Then
               tblKonyvSzerzoi.Edit;
            tblKonyvSzerzoiIroAz.Value:=tblIroIroAz.Value;
            tblKonyvSzerzoi.Post;
         end;
    end;
end;

procedure TfrmKonyv.btnSzerzoTorleseClick(Sender: TObject);
begin
  inherited;
  DMKonyvtar.tblKonyvSzerzoi.Delete;
end;
```

Ez mind nagyon szép és jó, de ez csak azért van, mert még nem gondoltunk a lehetséges hibákra. Mi van akkor, ha egy könyvnél ugyanazt a szerzőt másodszorra is megadjuk? A SZERZO.DB-be nem írhatjuk be kétszer ugyanazt az *ISBN;IroAz* párost, hiszen ez egyedi kulcs. Természetesen ekkor az új (duplikált) rekord postázása sikertelen lesz. Mi ezt a hibát a *tblKonyvSzerzoi* tábla *OnPostError* eseményében fogjuk észlelni és lekezelni (lásd a következő pontban).

Mielőtt egy tábla egy rekordját letörölnénk, illene rákérdezni a szándékra: *Biztosan le szeretné törölni?* Ezt is a következő pontban fogjuk megoldani.

Mint látjuk, vannak még megoldatlan kérdések. Az adatelérési komponensek eseményeinek köszönhetően (*OnPostError*, *OnDeleteError* stb.), megtehetjük azt, hogy fejlesztés közben csak a normál programlogikát követjük (esetleg feljegyezzük a felmerülő hibalehetőségeket). Később, miután az alkalmazás durva megoldása elkészült, átlépünk az adatmodulra, és ott minden előforduló hibát a megfelelő módon lekezelünk. (Ezzel foglalkozunk a következő pontban.)

Ahhoz, hogy egy könyv új adatait sikeresen felvihessük, még szükséges a következő kódrészlet.

```
procedure TfrmKonyv.RacsKarbantartasEnter(Sender: TObject);
begin
   inherited;
   If DMKonyvtar.tblKonyv.State in [dsInsert, dsEdit] Then
      DMKonyvtar.tblKonyv.Post;
end;
```

Amint a metódus nevéből is látjuk, ez akkor fog lefutni, amikor a szerzőket megjelenítő rács fókuszba kerül. Szerepe az, hogy a nyilvántartásunkba frissen felvett könyv adatait lementse (elpostázza) a KONYV.DB állományba. Ez azért fontos, mert a hivatkozási integritási feltétel miatt a SZERZO.DB-be mindaddig nem vihetjük fel egy könyv szerzőit, amíg azt a könyvet nem vesszük fel a KONYV.DB-be.

A *Karbantartás* oldallal ezennel meg is lennénk, nézzük a **cím szerinti keresést**. Minden fontosat öröklünk az *frmKarbanKeres* űrlapmintából, nekünk csak a *KeresesMezo* értéket kell ezúttal *'Cim'*-re beállítanunk.

```
procedure TfrmKonyv.FormCreate(Sender: TObject);
begin
   inherited;
   KeresesMezo:='Cim';
end;
```

A témakörök szerinti keresésre gondolva (ezt később, a 12. fejezetben fogjuk megírni), hozzunk létre egy új oldalt az *frmKonyv*-ben a *'Keresés téma szerint'* felirattal. Az új lapot hagyjuk egyelőre üresen.

Próbáljuk ki a *könyvek* űrlapot. Ehhez biztosítanunk kell a megjelenítését a főűrlapról. Akárcsak az íróknál, itt is állítsuk be a menüpontok *Tag* jellemzőjét a 0, 1 és 2 értékekre, majd kódoljuk le ezeket.

Komponens (Könyvek/~ menüpont)	Jellemző	Érték
Karbantartás	Tag	0
	OnClick	KonyvClick
Keresés Cím szerint	Tag	1
	OnClick	KonyvClick
Keresés Témakör szerint	Tag	2
	OnClick	KonyvClick
Nyomtatás	OnClick	KonyvNyomtatasClick

```
procedure TfrmFo.KonyvClick(Sender: TObject);
begin
  With frmKonyv Do
    begin
      Oldalak.ActivePage:= Oldalak.Pages[(Sender as TMenuitem).Tag];
      ShowModal;
    end;
end;

procedure TfrmFo.KonyvNyomtatas(Sender: TObject);
begin
  frmKonyv.btnNyomtatasClick(Sender);
end;
```

Próbálja ki az alkalmazást eddigi állapotában! Még nem kérdez vissza törlés előtt, nem figyeli a kötelezően kitöltendő mezőket stb. Elég sok minden nem az igazi még benne, de már határozottan közelítünk a vége felé.

10.5.3 Hibakezelés

Eddig egyértelműen eldőlt, hogy hány táblakomponenssel dolgozunk, és az is, hogy melyiket mire használjuk. Itt az ideje, hogy „bombabiztossá" tegyük alkalmazásunkat. Készítsünk egy táblázatot a lekezelendő problémákkal. Beleértjük ezekbe az esetleges törlés előtti visszakérdezéseket, és természetesen a tényleges hibának számító meghiúsult mentéseket és törléseket. El kell döntenünk, hogy mely tábláknál fordulhatnak egyáltalán elő ezek a problémák, és azt is, hogy konkrétan hogyan fogjuk orvosolni ezeket. Mindezeket egy táblázatban foglaljuk össze. A táblázatos megjelenítés azért jó, mert így biztosak lehetünk abban, hogy semmi nem kerüli el a figyelmünket, és sokkal inkább kézben tartható az egész alkalmazás hibakezelése.

Vegyük sorba a hibákat!

A törlés előtti visszakérdezés implementálásának legbiztosabb módja a megfelelő táblakomponens ***BeforeDelete*** eseményében van (lásd a 8. fejezet 8.4.4. és 8.4.7. pontjaiban). Így teljesen mindegy, honnan kezdeményezzük a törlést: a navigátorsorból, billentyűzetről vagy akár saját gombról. A *tblIroKonyvei* táblánk tartalmát csak megjelenítjük, sosem szerkesztjük át, így ebből törölni se fogunk, és az ő esetében mentési hibák sem állhatnak elő. A többi táblából törölhetünk rekordokat, így a visszakérdezést mindegyiküknél le kell kódolnunk.

A törlési kísérlet meghiúsulása az egy-több kapcsolatban álló táblák közül az egyes oldalinál fordulhat elő: könyvek, írók, témák és kiadók. Ezt a hibát a tábla ***OnDeleteError*** eseményében fogjuk észrevenni, és orvosolni. Ha a felhasználó le szeretne törölni például egy írót, akkor figyelmeztessük, hogy ez mindaddig lehetetlen, amíg ennek az írónak vannak könyvtárunkban könyvei. Ha viszont ki akar selejtezni egy könyvet, akkor ezt tegyük lehetővé úgy, hogy kitöröljük a SZERZO.DB-ből a kapcsolódó információkat is. Ezt is az

OnDeleteError-ban kell megoldanunk. A téma és kiadó törlésekor is hibaüzenetet jelenítünk majd meg.

Táblakomponens	Törlés előtti visszakérdezés (BeforeDelete)	Meghiúsult törlési kísérlet (OnDeleteError)	Postázáskor előállható hibák (OnPostError)	
			Kötelezően kitöltendő mező hiányzik	Elsődleges kulcs duplikálása
tblKonyv	✓	✓ (Egy könyv törlésekor ki kell törölnünk a szerzőit is a SZERZO.DB-ből)	✓ (Ki: 'Kérem, töltse ki az ISBN, Cím, Téma, Kiadó mezőket')	✓ (Ki: 'Ez az ISBN szám már szerepel a nyilvántartásunkban.')
tblIro	✓	✓ (Ki: 'Ez az író nem törölhető, mert vannak hozzátartozó könyvek.)	✓ (Ki: 'Kérem, töltse ki a Név mezőt')	–
tblKonyvSzerzoi	✓	–	–	✓ (Ki: 'Ez a szerző már szerepel a nyilvántartásunkban')
tblIroKonyvei	–	–	–	–
tblTema	✓	✓ (Ki:'Ez a témakör nem törölhető, mert vannak hozzátartozó könyvek.')	✓ (Ki: 'Kérem, töltse ki a Témakör mezőt')	–
tblKiado	✓	✓ (Ki: 'Ez a kiadó nem törölhető, mert vannak hozzátartozó könyvek.')	✓ (Ki: 'Kérem, töltse ki a Kiadó mezőt')	–

Mentési hibák két okból keletkezhetnek: ha a felhasználó nem tölt ki kötelezően kitöltendő mezőt (-ket) (például nem adta meg a könyv címét), vagy ha kulcsismétléseket idéz elő (például egy könyvnél ugyanazt a szerzőt kétszer akarná megadni).
Az első esetben egy üzenettel fel kell kérnünk a felhasználót a hiányzó adatok pótlására. Saját üzenetünk csak akkor fog megjelenni, pontosabban mondva az ***OnPostError*** esemény csak akkor fog bekövetkezni, ha a kötelezően kitöltendő mezők ***Required*** jellemzőjét *False*-ra állítjuk. E jellemző kezdőértéke az adatbázisban beállítottakból származik. Ha mi nem állítjuk át kézzel *False*-ra, akkor a Delphi hamarabb veszi észre a kötelezően kitöltendő mező hiányát, és az ő angol üzenetét jeleníti meg (ezt történt eddig, most itt az ideje, hogy változtassunk).

Kulcsismétlésekre is hibaüzenettel reagálunk. Ilyen hiba csak a *tblKonyv* és *tblKonyvSzerzoi* táblákban fordulhat elő. A *tblIro, tblTema* és *tblKiado* táblákban az azonosítót a program generálja, így ezekben biztosan nem lesznek ismétlések.

A hibatáblázat kitöltése után vegyük elő a fejlesztés során összegyűjtött hibalistát, és hasonlítsuk ezt össze a táblázattal. Ha valamiről megfeledkeztünk volna, akkor egészítsük ki a táblázatot.

Ezek után következhet a kivitelezés. Először is **keressünk közös elemeket**. Azt tapasztaljuk, hogy a törlés előtti visszakérdezést egy tábla kivételével mindenhol implementálnunk kell. Írjunk tehát erre **egy** metódust, és ezt hívjuk meg a táblakomponensek *BeforeDelete* eseményében.
A kötelezően kitöltendő mezőknél azonos a szöveg, csak a mezőnevek különböznek. Vezessünk be tehát egy konstanst *('Kérem töltse ki a(z) %s mezőt(-ket).')*. A konkrét hiba beálltakor a %s helyébe behelyettesítjük a konkrét mező nevét. Ugyanezt tesszük majd a törlési hibáknál is.

Törekedjünk arra, hogy hibaüzeneteink azonos hangvételűek legyenek. Ennek legjobb módja az, hogy összegyűjtjük, és egyszerre kezeljük le az előforduló hibákat.

Íme a kód:

```
unit UDMKonyvtar;

interface
  uses  Windows, Messages, SysUtils, Classes, Graphics...;
type
  TDMKonyvtar = class(TDataModule)
     ...
     end;

var
  DMKonyvtar: TDMKonyvtar;

const
  {Felvesszük konstansként az előforduló adatbázis-hibakódokat.
  Az OnPostError és OnDeleteError eseményekben ezen konstansok
  lekérdezésével fogjuk a hiba okát megtudni. A hibakódok teljes
  listája Delphi 2-esben és 3-asban a DELPHI\DOC\BDE.INT állományban
  található, Delphi 1-esben pedig a DELPHI\DOC\DBIERRS.INT-ben.}
  eKeyViol = 9729;
  eRequiredFieldMissing = 9732;
  eDetailRecordsExist = 9734;
```

```
   {üzenetek}
   TorlesJovahagyas = 'Biztos törölni kívánja?';
   KotelezoMezo = 'Kérem, töltse ki a(z) %s mezőt(-ket).';
   KulcsIsmetles = 'Ez a(z) %s már szerepel a nyilvántartásunkban.';
   NemTorolheto = 'Ez a(z) %s nem törölhető, mert vannak hozzátartozó'+
                  ' könyvek a nyilvántartásunkban';

implementation
{$R *.DFM}


{******************** általános ******************}
{Minden tábla BeforeDelete eseménye erre van irányítva}}
procedure TDMKonyvtar.BeforeDelete(DataSet: TDataSet);
begin
   If MessageDlg(TorlesJovahagyas, mtConfirmation,[mbYes, mbNo], 0) =
                                                        mrNo Then
      Abort; {ezzel szakítjuk meg a már beindult törlési folyamatot}
end;


{*****************  tblKonyv ************************}
procedure TDMKonyvtar.tblKonyvPostError(DataSet: TDataSet;
            E: EDatabaseError; var Action: TDataAction);
var Hibakod:Integer;
begin
   If E is EDBEngineError Then
      begin
         Hibakod:=(E as EDBEngineError).Errors[0].Errorcode;
         Case HibaKod Of
            eKeyViol:
               begin
                  ShowMessage(Format(KulcsIsmetles,['ISBN szám']));
                  Abort; {Ne jelenjen meg az angol hibaüzenet}
               end;
            eRequiredFieldMissing:
               begin
                  ShowMessage(Format(KotelezoMezo,['ISBN, Cím, Kiadó és'+
                                                   'Téma']));
                  Abort;
               end
         end;
      end;
end;
```

```
procedure TDMKonyvtar.tblKonyvDeleteError(DataSet: TDataSet;
                E: EDatabaseError; var Action: TDataAction);
begin
  {Bekövetkezik akkor, ha le akarjuk törölni, de nem lehet, mert
  vannak szerzői a SZERZO.DB-ben}
  If E is EDBEngineError Then
    If (E as EDBEngineError).Errors[0].Errorcode =
                                        eDetailRecordsExist Then
      begin
        {Töröljük le a szerzőket. Ahhoz, hogy ne kérdezzen vissza
        minden egyes szerzőnél, ideiglenesen lekapcsoljuk a
        BeforeDelete eseményt}
        With DMKonyvtar.tblKonyvSzerzoi Do
          begin
            BeforeDelete:=Nil;
            While RecordCount>0 Do
              Delete;
            BeforeDelete:= DMKonyvtar.BeforeDelete;
          end;
        {Próbáljuk újra a könyv kitörlését}
        Action:= daRetry;
      end;
end;


{***************** tblKonyvSzerzoi *****************}
procedure TDMKonyvtar.tblKonyvSzerzoiPostError(DataSet: TDataSet;
                  E: EDatabaseError; var Action: TDataAction);
begin
  {Bekövetkezik akkor, amikor például másodszor is be akarjuk állítani
  ugyanazt a szerzőt ugyanannál a könyvnél}
  If E is EDBEngineError Then
    If (E as EDBEngineError).Errors[0].Errorcode = eKeyViol Then
      begin
        ShowMessage(Format(KulcsIsmetles,['Könvy-író társítás']));
        DataSet.Cancel; {a már felvitt rekordot visszavonjuk}
        Abort;
      end;
end;


{***************** tblIro ********************}
procedure TDMKonyvtar.tblIroDeleteError(DataSet: TDataSet;
              E: EDatabaseError; var Action: TDataAction);
begin
  {Bekövetkezik akkor, ha le akarjuk törölni, de nem lehet, mert
  vannak könyvei a SZERZO.DB-ben}
```

```
   If E is EDBEngineError Then
      If (E as EDBEngineError).Errors[0].Errorcode =
                                          eDetailRecordsExist Then
         begin
            ShowMessage(Format(NemTorolheto,['író']));
            Abort;
         end;
end;

procedure TDMKonyvtar.tblIroPostError(DataSet: TDataSet;
                E: EDatabaseError; var Action: TDataAction);
begin
   If E is EDBEngineError Then
      If (E as EDBEngineError).Errors[0].Errorcode =
                                          eRequiredFieldMissing Then
         begin
            ShowMessage(Format(KotelezoMezo,['Név']));
            Abort;
         end;
end;


{****************** tblKiado ********************}
procedure TDMKonyvtar.tblKiadoDeleteError(DataSet: TDataSet;
                  E: EDatabaseError; var Action: TDataAction);
begin
   {Bekövetkezik akkor, ha le akarjuk törölni, de nem lehet, mert
   vannak könyvei a Konyv.DB-ben}
   If E is EDBEngineError Then
      If (E as EDBEngineError).Errors[0].Errorcode =
                                 eDetailRecordsExist Then
         begin
            ShowMessage(Format(NemTorolheto,['kiadó']));
            Abort;
         end;
end;

procedure TDMKonyvtar.tblKiadoPostError(DataSet: TDataSet;
                E: EDatabaseError; var Action: TDataAction);
begin
   If E is EDBEngineError Then
      If (E as EDBEngineError).Errors[0].Errorcode =
                                 eRequiredFieldMissing Then
         begin
            ShowMessage(Format(KotelezoMezo,['Kiadó']));
            Abort;
```

```
      end;
end;


{****************** tblTema ********************}
procedure TDMKonyvtar.tblTemaDeleteError(DataSet: TDataSet;
                    E: EDatabaseError; var Action: TDataAction);

begin
  {Bekövetkezik akkor, ha le akarjuk törölni, de nem lehet, mert
  vannak könyvei a KONYV.DB-ben}
  If E is EDBEngineError Then
    If (E as EDBEngineError).Errors[0].Errorcode =
                                eDetailRecordsExist Then
      begin
        ShowMessage(Format(NemTorolheto,['témakör']));
        Abort;
      end;
end;

procedure TDMKonyvtar.tblTemaPostError(DataSet: TDataSet;
                  E: EDatabaseError; var Action: TDataAction);
begin
  If E is EDBEngineError Then
    If (E as EDBEngineError).Errors[0].Errorcode =
                               eRequiredFieldMissing Then
      begin
        ShowMessage(Format(KotelezoMezo,['Téma']));
        Abort;
      end;
end;

{****************** tblKonyvSzerzoi ********************}
{Ez azért szükséges, hogy amikor a tblKonyvSzerzoi felől írunk a
SZERZO.DB-be, akkor a tblIroKonyvei „vegye észre" a változtatásokat.
A rutint a tblKonyvSzerzoi.AfterPost eseményére építjük.}
procedure TDMKonyvtar.tblIroKonyveiFrissitese(DataSet: TDataSet);
begin
  tblIroKonyvei.Refresh;
end;


{****************** táblák megnyitása ********************}
{Most már tervezési időben bezárjuk a táblákat, a megnyitásokat
futási időre halasztjuk. Figyelem a sorrendre! Ha egy tábla
egy másikból származtatott mezőt tartalmaz, akkor a forrástáblát
még a származtatott mezőt tartalmazó tábla előtt meg kell nyitnunk.}
```

```
procedure TDMKonyvtar.DMKonyvtarCreate(Sender: TObject);
begin
   tblTema.Open;
   tblKiado.Open;
   tblKonyv.Open;
   tblIroKereso.Open;
   tblKonyvSzerzoi.Open;
   tblIro.Open;
   tblIroKonyvei.Open;
end;

{****************** táblák bezárása *********************}
procedure TDMKonyvtar.DMKonyvtarDestroy(Sender: TObject);
begin
   tblTema.Close;
   tblKiado.Close;
   tblKonyv.Close;
   tblIroKereso.Close;
   tblKonyvSzerzoi.Close;
   tblIro.Close;
   tblIroKonyvei.Close;
end;

end.
```

Tesztelje le az alkalmazást! Most már remélhetőleg minden megfelelően működik a témakör szerinti keresés és a nyomtatások kivételével. Ezekkel későbbi fejezetekben foglalkozunk.

11. SQL utasítások a Delphiben

Az SQL (*Structured Query Language*) az adatbázisok kezelésére kidolgozott strukturált lekérdező nyelv. Delphi alkalmazásainkban is futtathatunk SQL utasításokat a *TQuery* komponens segítségével. Ebben a fejezetben megismerkedünk az SQL utasítások Delphiből történő használatának lehetőségeivel.

11.1 Az SQL és a BDE

Az SQL utasítások három kategóriába sorolhatók:

- **DDL** (*Data Definition Language*) utasítások, melyekkel az adatbázis szerkezetét módosíthatjuk (táblákat, nézeteket stb. hozhatunk létre, módosíthatunk és törölhetünk). Például *CREATE TABLE, CREATE VIEW, CREATE INDEX, DROP TABLE* stb.
- **DML** (*Data Manipulation Language*) utasítások, melyekkel az adatbázis adatait kezeljük (adatok szűrése, módosítása, törlése). Ezek a *SELECT, INSERT, UPDATE* és *DELETE* utasítások.
- **DCL** *(Data Control Language)* utasítások, melyekkel egyéb adatbázis-kezelési feladatokat láthatunk el (tranzakciók kezelése, jogosultságok beállítása, adatbázis karbantartása, mentése stb.). Például a *BEGIN TRANSACTION, COMMIT, ROLLBACK, GRANT, REVOKE stb.*

Az SQL utasításokat Delphiben a *TQuery* komponens segítségével adhatjuk meg, végrehajtásukat pedig a BDE irányítja: ha az SQL parancs adatbázis-szerveren levő adatokra vonatkozik, akkor ezt az adatbázismotor legtöbbször egy-az-egybe továbbítja a szerver felé (PASSTHRU SQL). Ez esetben az utasítást a szerver hajtja végre, így szintaxisának ehhez kell igazodnia.

Vannak azonban olyan esetek is, amikor a BDE magához ragadja az SQL végrehajtásának jogát. Például tegyük fel, hogy egy SQL-el több fizikai adatbázisból szeretnénk adatokat lekérdezni. Ezek az adatok csak a BDE szintjén futnak össze, így az SQL utasítást is itt kell végrehajtani. Ennek ellenére az álnév *SQLQRYMODE* opciójával ezt még befolyásolni lehet. Az *SQLPASSTHRU MODE* beállítással pedig az PASSTHRU SQL utasítások tranzakciószintű viselkedését finomíthatjuk. Bővebben lásd a Database Explorer súgójában.

Ha nem szerverhez intézünk egy SQL utasítást, akkor ezt a BDE helyileg hajtja végre. Ekkor már értelemszerűen nem minden SQL parancs használható. A megengedett utasítások halmaza a Delphi 1-es verziója után egyre bővült, a pontos szintaxist lásd a súgóban. Néhány, nem használható utasítás feladatát elláthatjuk Delphiből a különböző komponensek metódusaival: például a tranzakció-kezelést a *TDatabase* osztály *StartTransaction, Commit* és *Rollback* metódusaival valósíthatjuk meg (lásd 8. fejezet).

Ha egy lekérdező (*SELECT*) utasítást hajtunk így végre, akkor az eredményhalmazt a *TQuery* komponens fogja tartalmazni. Az adatok egy – a *TQuery*-re irányított – *TDataSource* és a megfelelő megjelenítési komponensek segítségével jeleníthetők meg. No, de kezdjük a legelején. Ismerkedjünk meg a *TQuery* komponenssel.

11.2 A TQuery komponens

- Helye az osztályhierarchiában: *TObject/TComponent/TDataset/... TDBDataSet/ TQuery.*
- Szerepe: egy SQL utasítás végrehajtását teszi lehetővé.
- Fontosabb jellemzői:

  Az előbbi fejezetekben az adathalmazokról ismertetett fogalmak és technikák, a *TQuery* komponensre is vonatkoznak (megnyitás, zárás, navigálás, keresés, szűrés, mezők...).

  - **DatabaseName**: az adatbázis-komponens vagy az álnév melyből az adatok származnak
  - **SQL:TStrings**: az SQL utasítás szöveges formában.

    ```
    Pl: Query1.SQL := 'SELECT * FROM Customer';
    ```

    Az SQL utasítás megadásánál több megoldás közül választhatunk: vagy begépeljük tervezéskor közvetlenül az *SQL* jellemzőbe, vagy valamilyen vizuális eszközzel építjük fel, majd a legenerált utasítást behozzuk az *SQL* jellemzőbe. Természetesen futásidőben kódból is felépíthetünk és lefuttathatunk lekérdezéseket. Mindezen módszereket a 11.4. pontban ismertetjük.
  - **DataSource**: csak paraméteres lekérdezések esetén van értelme. Olyankor a *DataSource* jellemzőjét arra az adatforrásra irányíthatjuk, amelyből a paraméter értéke származik. Bővebben lásd a 11.5. pontban.
  - **Params**: a lekérdezés paramétereinek tömbje (11.5. pont)
  - **RequestLive:Boolean**: a *SELECT* lekérdezés eredményhalmazának szerkeszthetőségét lehet vele beállítani. Alapértelmezés szerint *hamis* értékű. Ha *igazra* állítjuk, és ugyanakkor lekérdezésünk megfelel az SQL-92 szabvány által előírt szerkeszthetőségi szabályoknak, akkor az SQL utasítás eredményhalmaza szerkeszthető lesz. Ellenkező esetben az adatok csak olvashatók, és nem írhatók. Nézzük a legfontosabbakat a lekérdezés szerkeszthetőségének előfeltételeiből:
    - Egyetlen táblán alapuljon

- Ne használjon aggregáló függvényeket (*SUM*, *AVG* stb.)
- Ne tartalmazzon számított mezőt a *SELECT* listában...

• Fontosabb metódusai:

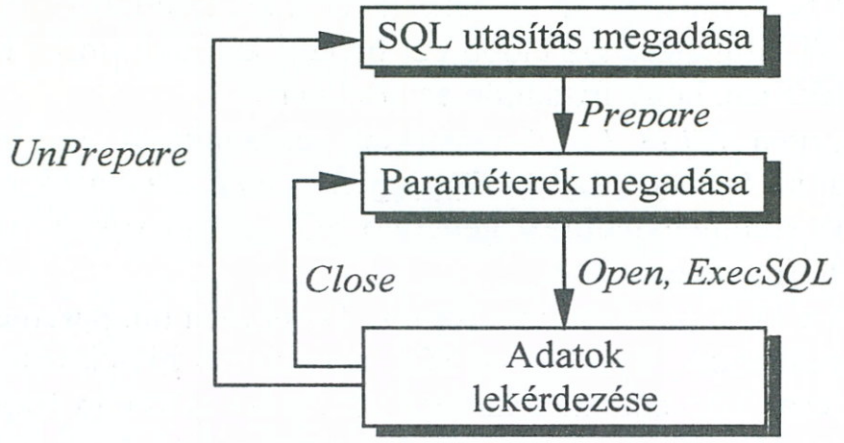
- **Open**: a *SELECT* SQL utasítást tartalmazó lekérdezés lefuttatását eredményezi. A lekérdezés újbóli lefuttatásához előbb ezt be kell zárnunk, majd újra lefuttathatjuk (annak érdekében, hogy frissüljön az eredmény).
- **ExecSQL**: egyéb, nem *SELECT* SQL utasítások végrehajtását teszi lehetővé. Ezt használjuk például az *INSERT*, *UPDATE* vagy *DELETE* utasítások esetében. Az *Open* metódust az eredményhalmazt generáló SQL parancsoknál alkalmazzuk, míg az *ExecSQL*-t minden egyébnél.
- **Prepare**: mint ezt a 11.5. pontban látni fogjuk, Delphiben ún. paraméteres lekérdezéseket is használhatunk. Ilyenkor az SQL utasításban hivatkozunk egy paraméterre, ez pedig csak közvetlenül a parancs lefuttatása előtt kap értéket (például le akarjuk kérdezni két dátum közötti tanfolyamainkat úgy, hogy a konkrét dátumhatárokat a felhasználó írja be) (bővebben lásd a 11.5. pontban). A *Prepare* metódus használata a többször lefuttatandó paraméteres lekérdezések esetében ajánlott. Egyszer kell meghívni, az SQL utasítás megadása után, de még mielőtt a paramétereknek konkrét értékeket adnánk. A paraméterek fogadására készíti fel az adatbázis-szervert, így az SQL parancs egymás utáni lefuttatásai (más és más paraméterekkel) gyorsabbak lesznek.

11.3 A TQuery komponens használata

Amikor egy lekérdezést szeretnénk alkalmazásunkban lefuttatni, akkor a következő lépéseket kell követnünk:

1. Helyezzünk el egy *TQuery* komponenst az adatmodulon vagy az űrlapon. Állítsuk be *Name* jellemzőjét egy, a szerepére utaló névre (például *q95osRendelesek*)
2. A *DatabaseName* jellemzőjében válasszuk ki a használt adatbázis-komponens nevét vagy az álnevet (például *DBDEMOS*).
3. Írjuk be az *SQL* jellemzőbe a megfelelő utasítást.
4. Ha a lekérdezés eredményét meg szeretnénk jeleníteni, akkor irányítsunk rá egy *DataSource* komponenst (*dsrq95osRendelesek*). Az adatmegjelenítési elemeket ehhez az adatforráshoz kell kapcsolnunk.
5. Ha a lekérdezés paramétereket tartalmaz, akkor hívjuk meg a *Prepare* metódusát.
6. Ha a lekérdezés paraméteres, akkor adjunk ezeknek értéket.
7. Futtassuk le a lekérdezést az *Open* vagy az *ExecSQL* utasítással attól függően, hogy generál-e vagy nem eredményhalmazt. A *SELECT* lekérdezéseket tervezési időben is végrehajthatjuk az *Active* jellemzőjük Igazra állításával.
8. Ha a lekérdezést újra le szeretnénk futtatni, akkor előbb zárjuk be, majd ismételjük meg a 6, 7, 8 lépéseket.

9. Ha az SQL utasítást meg szeretnénk változtatni, akkor előbb érdemes az *UnPrepare* metódussal felszabadítani a *Prepare* hívásával lefoglalt területet (a lekérdezés előkészítésének céljából).



11.1. ábra. A *TQuery* komponens használata

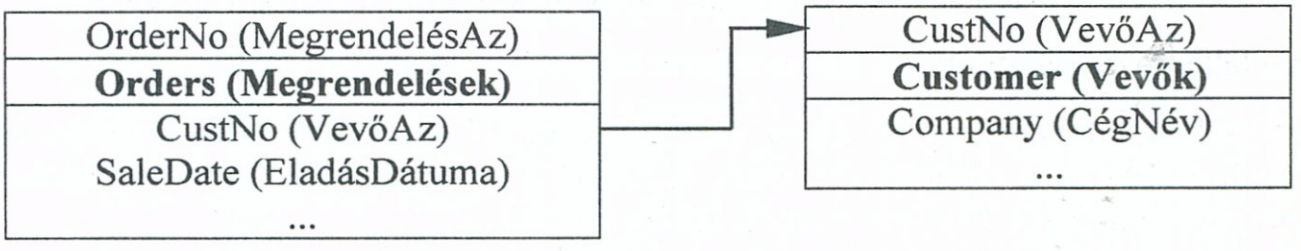
11.4 Az SQL utasítás megadásának módozatai

A *TQuery* komponens SQL jellemzőjében nem hivatkozhatunk Delphi komponensnevekre: sem táblákra, sem mezőkre, sem szerkesztődobozokra stb. Hallgatóim gyakran hajlamosak például arra, hogy az SQL utasításba belefoglalják a – csak Delphiben létező – származtatott mezőket. Az SQL parancs csak a fizikai táblákat és ezek fizikailag létező mezőit kérdezheti le.

Feladat: a különböző módozatok bemutatására készítsünk egy alkalmazást három lekérdezéssel:

- 1995. január 1. utáni megrendelések: MegrendelésAz, VevőAz, EladásDátum.
- 1995. január 1. utáni megrendelések: MegrendelésAz, VevőNév, EladásDátum.
- 1995. január 1. óta melyik vevő hányszor vásárolt: VevőAz, VevőNév, VásárlásokSzáma.

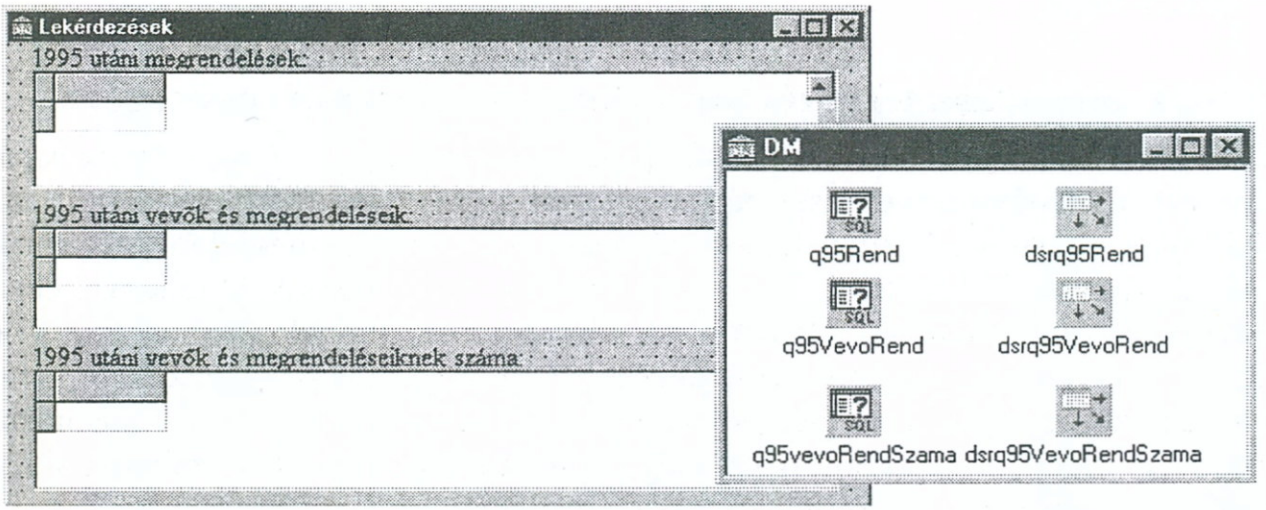
A lekérdezéseket a mintaadatbázison fogjuk lefuttatni az *Orders* és a *Customer* táblákon. Ezeket Delphiből a *DBDEMOS* álnévvel érhetjük el.



11.2. ábra. A használt táblák

Megoldás (🖫 11_LEKERDEZES\PQUERY.DPR)

Kezdjünk egy új alkalmazást, benne hozzunk létre egy adatmodult. Helyezzünk el rajta három *TQuery* komponenst: *q95Rend*, *q95VevoRend*, *q95VevoRendSzama*. Állítsuk be *DatabaseName* jellemzőjüket *DBDEMOS*-ra. Helyezzünk el az adatmodulra három *DataSource* komponenst is. Az alkalmazás űrlapján jelenítsük meg a lekérdezések eredményét három rácsban. Körülbelül így:



11.3. ábra. Lekérdezéses mini-alkalmazásunk űrlapja és adatmodulja

11.4.1 SQL megadása tervezéskor begépeléssel

Ez a leggyorsabb módszer, persze csak akkor, ha jól ismerjük az SQL szintaxist. Kattintsunk az objektum-felügyelőben az *SQL* jellemző melletti gombra. A megjelenő ablakba máris begépelhetjük a megfelelő SQL utasítást. A három utasítás a következő:

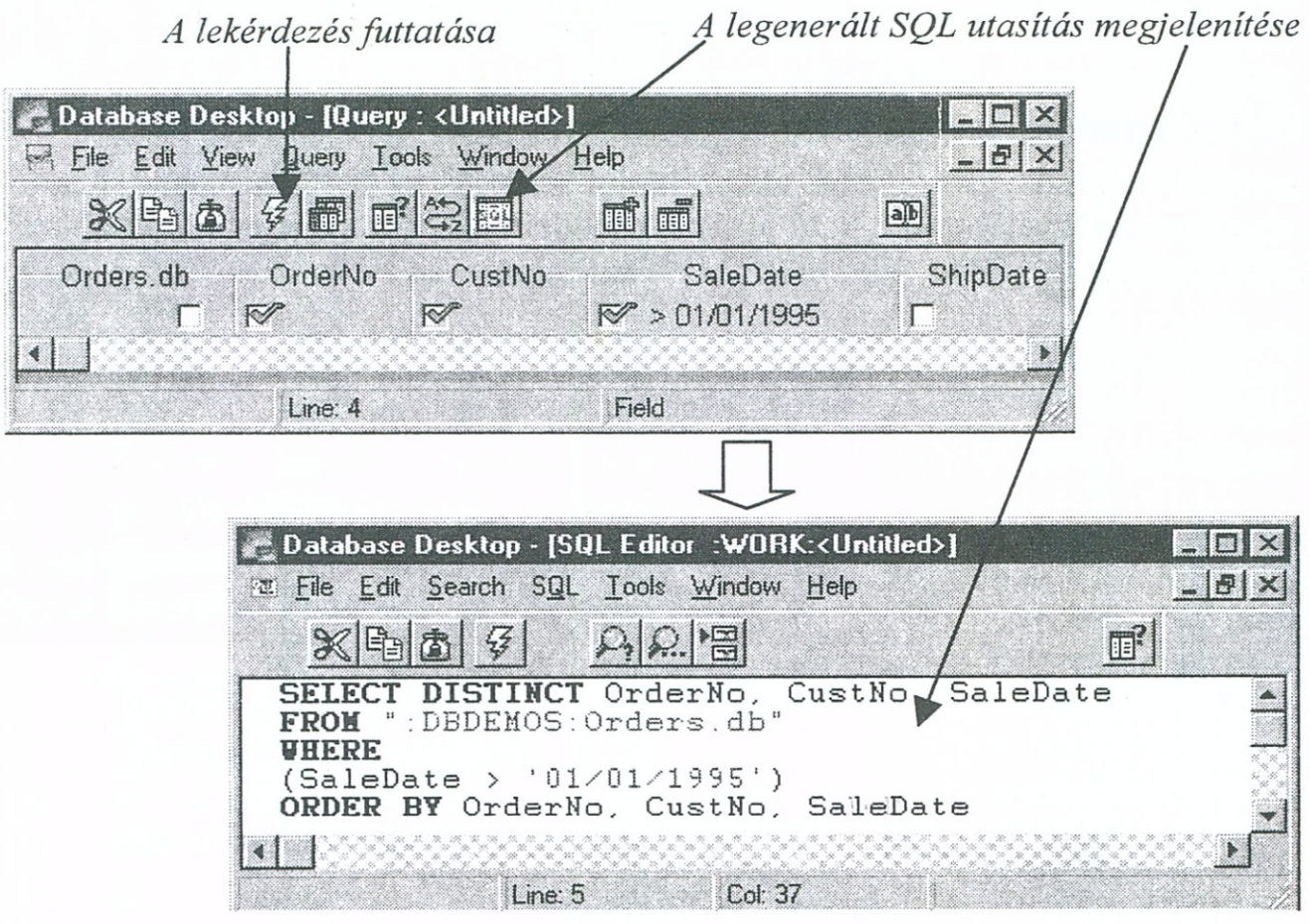
```
{q95Rend.SQL}
SELECT OrderNo, CustNo, SaleDate
FROM Orders
WHERE SaleDate > '01/01/1995'

{q95VevoRend.SQL}
SELECT OrderNo, Company, SaleDate
FROM Orders, Customer
WHERE   (Orders.CustNo = Customer.CustNo)
      AND (SaleDate > '01/01/1995')

{q95VevoRendSzama.SQL}
SELECT CustNo, Company, Count(OrderNo)
FROM Orders, Customer
WHERE   (Orders.CustNo = Customer.CustNo)
      AND (SaleDate > '01/01/1995')
GROUP BY CustNo, Company
```

A fenti lekérdezés „szó szerint" a következőképpen értelmezendő: „...csoportosítsd *VevőAz* szerint, és azon belül *VevőNév* szerint...". Ennek így logikailag semmi értelme (hiszen egy *VevőAz* értékű csoportban eleve csak egy rekord van), de technikailag **minden aggregációban részt nem vevő és ugyanakkor megjelenített mező nevét a *GROUP BY*–ban is fel kell tüntetnünk.**

Nyissuk meg a lekérdezéseket *(Active := True)*. Hasonlítsuk össze a három rács eredményét.

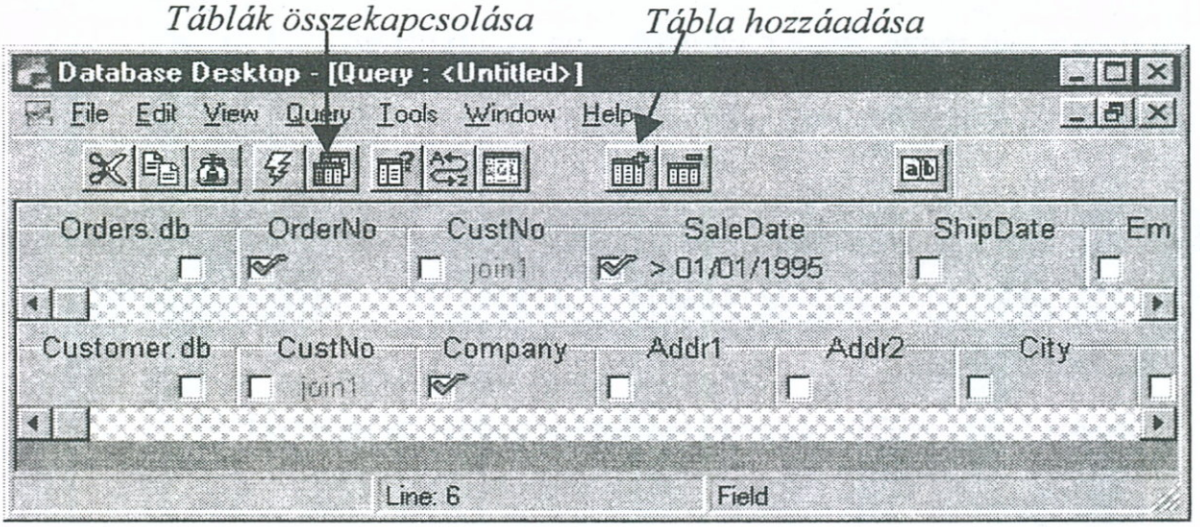
11.4.2 SQL megadása tervezéskor a Database Desktop segítségével

Építsük fel a lekérdezést a Database Desktop-ban a vizuális QBE (*Query by Example*) rács segítségével. A vizuális tervezéssel párhuzamosan a Database Desktop felépíti az ennek megfelelő SQL utasítást. Miután lekérdezésünket leteszteltük, megjelenítjük annak SQL utasítását, és vágólapon keresztül behozzuk a *TQuery* komponens *SQL* jellemzőjébe.
Ehhez indítsuk el a Database Desktop segédprogramot. Hívjuk meg a *File/New/QBE Query...* menüparancsát. **Az első lekérdezést** az *Orders* táblán végezzük, így egyelőre elég, ha ezt választjuk ki a megjelenő listából. A QBE ablakban máris felsorakoztak a tábla mezői. A megjelenítendő mezőket elég bepipálni. Ha feltételt is szabunk, akkor azt a megfelelő mező neve alá kell írnunk.



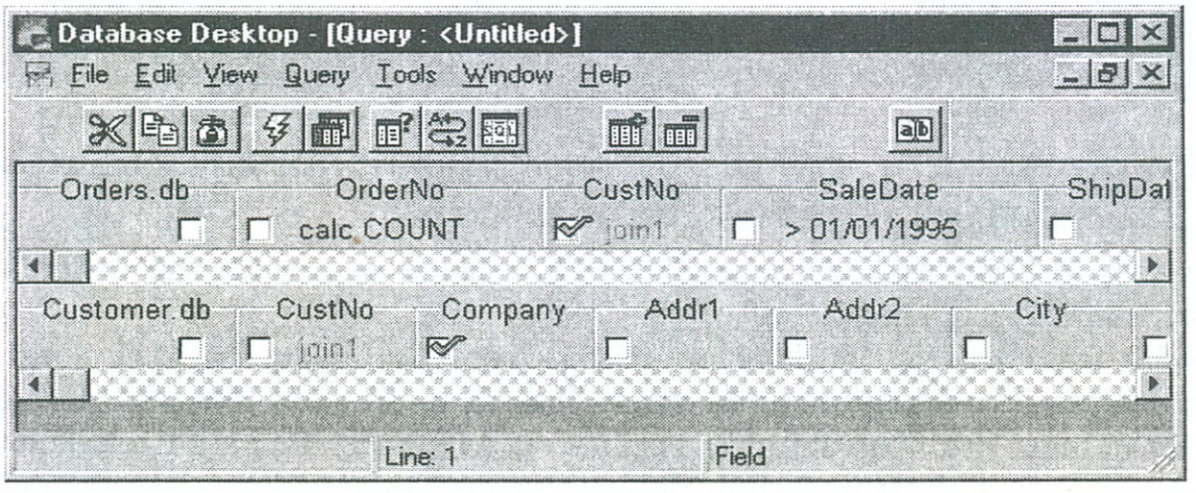
11.4. ábra. Az 1995. január 1. utáni megrendelések lekérdezése a Database Desktop-ban

Nézzük **a második lekérdezést**. Ehhez már két tábla szükséges: az *Orders* tábla a *SaleDate (EladásDátum)* mezője miatt, a *Customer* pedig a *Company (VevőNév)* miatt. Az előző lekérdezést folytatva, olvassuk be a *Customer* táblát is (11.5. ábra). Ha már több táblánk van, akkor ezeket össze is kell kapcsolnunk. Kattintsunk a *'Táblák összekapcsolása'* gombra, majd a megváltozott egérkurzorral kattintsunk a két összekötő mezőbe *(CustNo–CustNo)*. A megjelent piros szöveg *(join1)* jelzi a kapcsolatot. Ugyanezt a hatást érjük el akkor is, ha a megfelelő mezőknél az F5 billentyű lenyomása után írjuk be a szöveget. A szöveg tartalma nem számít; a lényeg az, hogy azonos legyen mindkét kapcsolódó mezőnél.



11.5. ábra. Az 1995 utáni vevők és megrendeléseik

A **harmadik lekérdezés**nél csoportosítanunk kell a megrendeléseket *VevőAz* szerint, és minden csoportban össze kell ezeket számolnunk. Ezt a *Count* függvénnyel fogjuk megvalósítani. Figyelem, csak azokat a mezőket pipáljuk ki, amelyek a csoportosításban is részt vesznek. A mi esetünkben ez a *CustNo* és *Company*. A *Company* a megjelenítésért fontos. Ha a *SaleDate* mezőt is kipipálnánk, akkor a csoportosítás vevő és azon belül eladási dátum szerint történne, így a vevők rendeléseinek számát napi bontásban kapnánk meg.



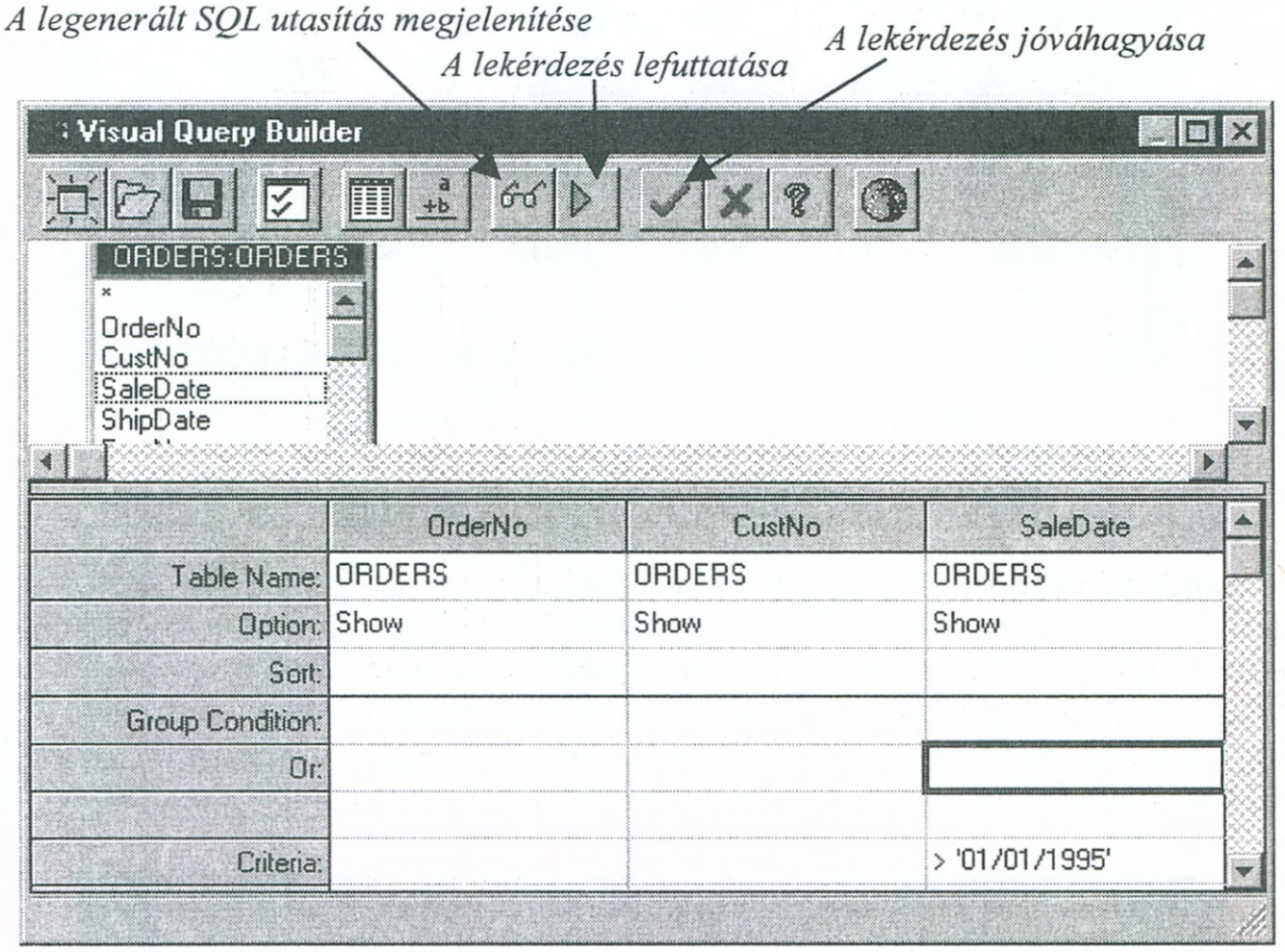
11.6. ábra. Az aggregáló függvényeket a *'calc'* szó után kell beírnunk

Próbálja ki a Database Desktop-ban legenerált lekérdezéseket.

11.4.3 SQL megadása a vizuális szerkesztővel (*Visual Query Builder*)

A 32 bites Delphi Client/Server változataiban rendelkezésünkre áll egy beépített vizuális lekérdezés-készítőt (*Visual Query Builder*).

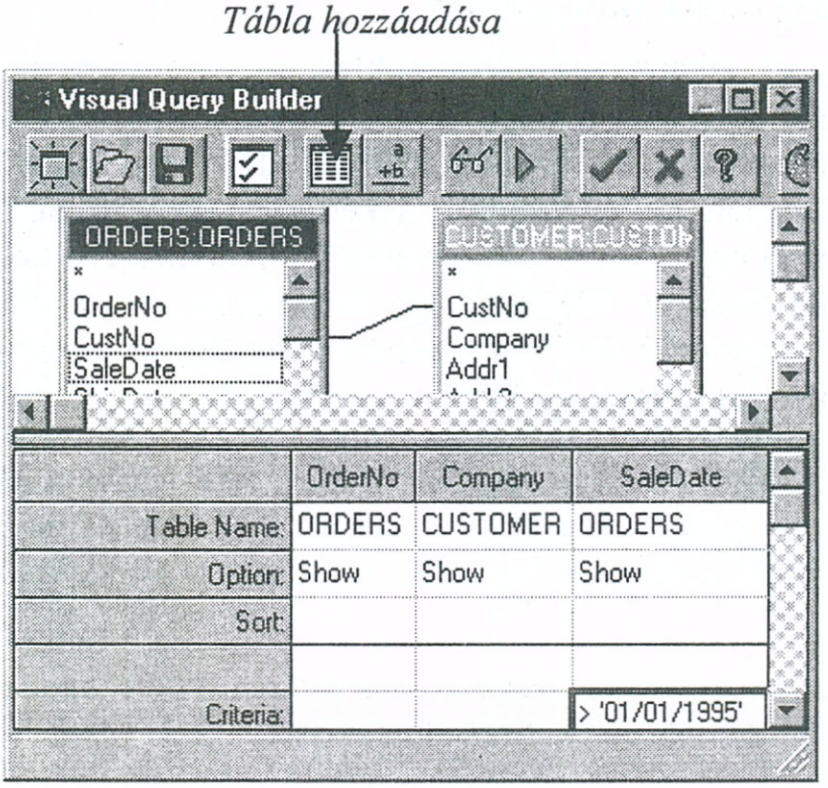
Nézzük az **első lekérdezést**: jelöljük ki a *TQuery* komponensét, majd a gyorsmenüből hívjuk meg a *Query Builder...* parancsot. Itt is először ki kell választanunk a lekérdezendő táblát (-kat). Ezek után a megjelenítendő mezőket vonszoljuk le az ablak alsó felébe. A feltételt a megfelelő mező alá, a *Criteria* sorba kell beírnunk.



11.7. ábra. Az 1995.01.01. utáni megrendelések a beépített vizuális lekérdezés-készítővel

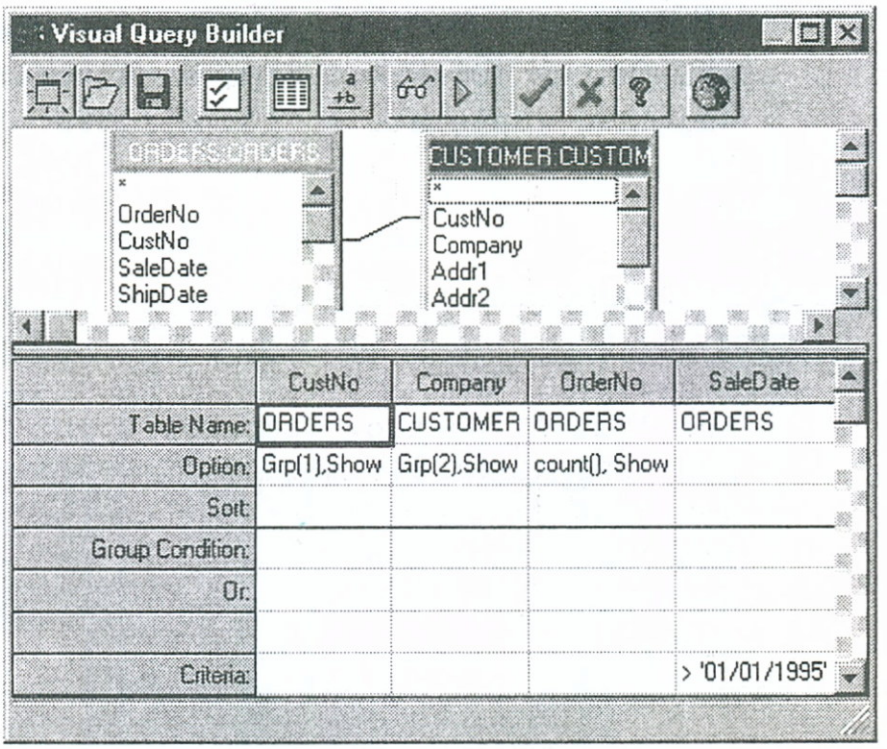
A lekérdezés jóváhagyásakor a legenerált SQL utasítás automatikusan bekerül a *TQuery* komponensünk *SQL* jellemzőjébe.

A **második lekérdezés**nél szükségünk van a *Customer* táblára is (11.8. ábra). A táblák összekapcsolása itt a legegyszerűbb: vonszoljuk az egyik tábla *CustNo* mezőjét a másik tábla *CustNo* mezőjére. (Ugyanígy van például az MS Accessben is). Ezek után töröljük ki a *CustNo* mezőt az alsó ablakrészből. Helyette a *Company*-t kell megjelenítenünk.



11.8. ábra. Az 1995.01.01. utáni vevők és megrendeléseik
(A táblákat itt vonszolással kapcsoljuk össze)

A **harmadik lekérdezés**hez hozzuk le a *CustNo* mezőt. Tanulmányozzuk át az *Options* sor gyorsmenüjét: itt állítható be, hogy a mező látható legyen-e vagy sem (a *SaleDate* mező nem látható, csak azért hoztuk le, hogy feltételt szabjunk rá). Ugyancsak itt állítjuk be a *CustNo,* és a *Company* szerinti csoportosítást is. Az *OrderNo* oszlop rekordjait csoportonként össze kell számolnunk. Ezt a *Count* függvénnyel valósítjuk meg.



11.9. ábra. Az 1995. január 1. utáni megrendelések száma vevőnként csoportosítva

11.4.4 Az SQL megadása futásidőben

Ha azt szeretnénk, hogy alkalmazásunkban sokféle szempont szerint lehessen leválogatásokat készíteni, akkor a konkrét szempontok alapján futáskor egy dinamikus SQL lekérdezést kell összeállítanunk. A felhasználó bejelöli a feltételben szereplő mezőket, megadja értékeiket, és ezek alapján mi felépítjük a lekérdezést képező SQL utasítást. Ez egy karakterlánc lesz, melyet egy *TQuery* komponens *SQL* jellemzőjébe kell elhelyeznünk.
Nézzük, hogyan lehet futáskor felépíteni egy lekérdezést. Az előző három lekérdezés létrehozását most az adatmodul *OnCreate* eseményébe írjuk, de egy éles alkalmazásban természetesen máshol lenne a helye (például a *Leválogatás* gomb kattintására):

```
procedure TDM.DMCreate(Sender: TObject);
begin
   With q95Rend,SQL Do[1]
      begin
         Clear;
         Add('SELECT OrderNo, CustNo, SaleDate FROM Orders');
         Add('WHERE SaleDate > ''01/01/1995''');
         Open;
      end;

   With q95VevoRend,SQL Do
      begin
         Clear;
         Add('SELECT OrderNo, Company, SaleDate FROM Orders,Customer');
         Add('WHERE (Orders.CustNo = Customer.CustNo)');
         Add('AND (SaleDate > ''01/01/1995'')');
         Open;
      end;

   With q95VevoRendSzama,SQL Do
      begin
         Clear;
         Add('SELECT CustNo, Company, Count(OrderNo)');
         Add('FROM Orders,Customer');
         Add('WHERE (Orders.CustNo = Customer.CustNo)');
         Add('AND (SaleDate > ''01/01/1995'')');
         Add('Group By CustNo, Company');
         Open;
      end;
end;
```

A lekérdezést egy szöveges állományból is betölthetjük a *LoadFromFile* metódussal:

---

[1] Itt a *With* utasításban két, vesszővel elválasztott változóval minősítünk: *With q95Rend, SQL*. Így a minősítés értelemszerűen vagy a *q95Rend*, vagy a *q95Rend.SQL*-lel fog megtörténni: *q95Rend.SQL.Clear, q95Rend.SQL.Add(...)... q95Rend.Open*.

```
q95Rend.SQL.LoadFromFile('Lekerdezes.SQL');
q95Rend.Open;
```

11.5 Paraméteres lekérdezések

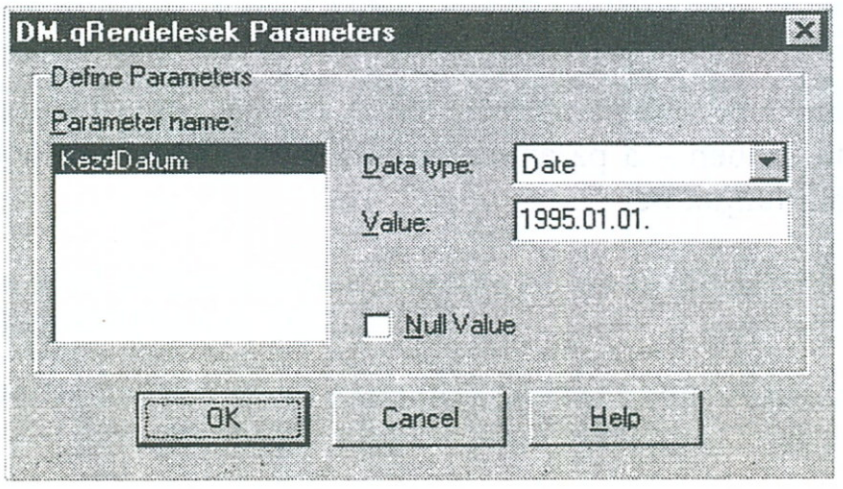
```
{qRendelesek.SQL}
SELECT * FROM Orders
WHERE SaleDate >= :KezdDatum
```

A paraméteres lekérdezések nagyobb rugalmasságot visznek alkalmazásainkba. Megfogalmazhatunk egy SQL utasítást úgy, hogy a változó részét paraméterként adjuk meg. A ':' jelzi, hogy a közvetlenül utána következő szöveg egy paraméter neve (*KezdDatum*). Természetesen a lekérdezést csak a paraméter értékének megadása után futtathatjuk majd le.

```
With qRendelesek Do
  begin
    Prepare;
    ...
    Params[0].AsDate:= StrToDate('1995.01.01');
    Open;
    ...
    Close;
    Params[0].AsDate:= StrToDate('1996.01.01');
    Open;
    ...
  end;
```

11.5.1 A paraméter (-ek) megadásának módozatai

- **Tervezéskor** a paraméterek értékét a ***Params*** jellemző segítségével adhatjuk meg. Ezt a módszert csak akkor szoktuk használni, amikor tervezési időben szintaktikailag le akarjuk tesztelni az SQL utasítást.



11.10. ábra. Paraméter beállítása tervezési időben (*Params* jellemző)

- Az igazi paraméterátadás **futásidőben** történik. Ekkor is használhatjuk a ***Params*** jellemzőt, amint az előző példában láttuk. Indexként a paraméter sorszámát kell megadnunk (az első paraméter indexe 0, a másodiké 1...). Itt is használhatók a mezőobjektumoknál már megszokott konverziós jellemzők (*AsString, AsDate...*).

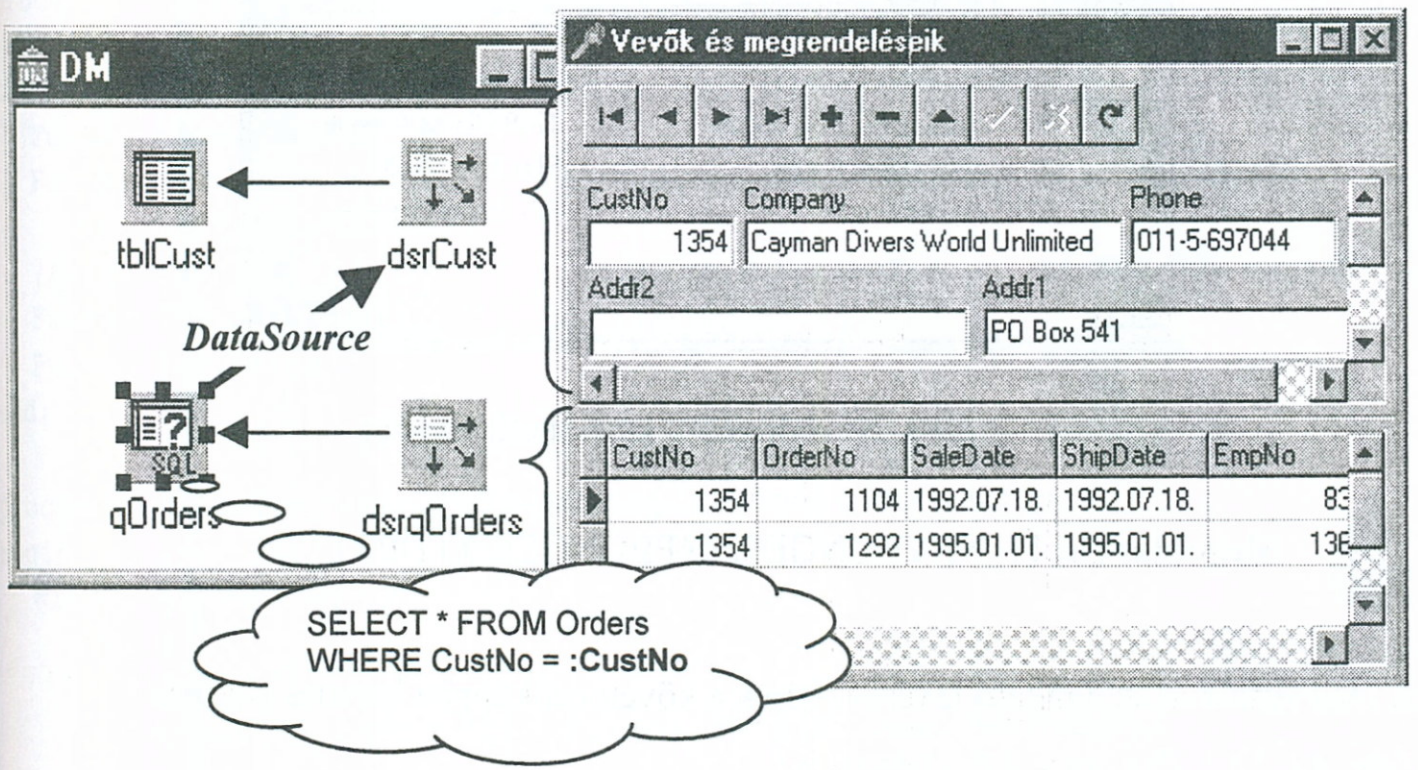
  Maradjunk az előbb említett példánál. A *qRendelesek* egy bizonyos dátum utáni megrendeléseket válogatja le. A felhasználó a dátumhatárt egy szerkesztődobozban adja meg. Ezután a lekérdezés paraméterét be kell állítanunk – a *Params* segítségével – a beírt értékre. Ez az esemény nem lehet a szerkesztődoboz *OnChange* eseménye, mivel ez bekövetkezik akkor is, amikor a felhasználó még csak a dátum egy részét írta be. El kell tehát helyeznünk egy külön gombot (*btnLekerdezes*). Figyelem a dátumkonverzióra!

```
{🖫 11_DATUMPARAM\PPARAM.DPR}
procedure TfrmRendelesek.btnLekerdezesClick(Sender: TObject);
begin
  With DM.qRendelesek Do
     begin
        Try
           Close;
           Params[0].AsDate:= StrToDate(eDatum.Text);
           Open;
        Except
           On EConvertError Do
              begin
                 {az üzenetet „megtörjük" a #13#10 karakterekkel}
                 ShowMessage('A szerkesztődobozba helyes dátumot'+
                             ' kell írnia! '+ #13#10 +
                             'Például: '+ DateToStr(Date));
                 Open;
              end;
        End;
     end;
end;
```

- Ha lekérdezésünknek több paramétere lenne, és nem ismerjük ezek sorrendjét, akkor – ugyancsak futásidőben – a paraméterértékeket a *TQuery* komponens ***ParamByName*** metódusával állíthatjuk be:

```
With qRendelesek Do
  begin
     Close;
     ParamByName('KezdDatum').AsDate:= StrToDate(Datum.Text);
     Open;
  end;
```

- Ha a paraméter értéke egy adatforrás valamely mezőjéből származik (és nem például a felhasználó gépeli ezt be), akkor a paraméterérték (-ek) megadására használhatjuk a *TQuery* komponens ***DataSource*** jellemzőjét. Ezt arra az adatforrásra kell irányítanunk, amelyikből a paraméter értéke származik. A lekérdezés lefuttatásakor a rendszer kiértékeli, hogy be van-e állítva a *DataSource* jellemzője, és ha igen, akkor az ott beállított adatforrásban keres egy – a paraméter nevével megegyező nevű – mezőt. Ennek a mezőnek aktuális rekordbeli értéke kerül a lekérdezésbe paraméterértékként. Később, amikor az aktuális rekord megváltozik, tehát a megfelelő mező értéke is más lesz, a rendszer automatikusan frissíti a lekérdezést, újrafuttatja az új paraméterértékkel.



11.11. ábra. Fő-segéd űrlap megvalósítása paraméteres lekérdezéssel

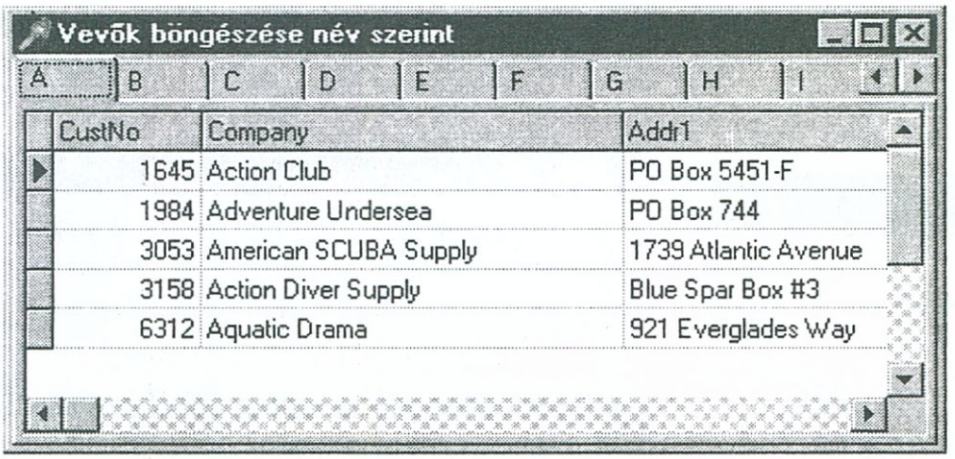
A fő-segéd űrlapok paraméteres lekérdezéssel is megvalósíthatók. A rácsban csak az aktuális vevő megrendeléseit szeretnénk látni. Ezért a segédűrlap tartalmát (a megrendeléseket) egy paraméteres lekérdezéssel válogatjuk le, melyben a paraméter értéke mindig a főűrlapon megjelenő vevő azonosítójával lesz egyenlő.

Készítse el a vevő-megrendelések űrlapot (11.11. ábra)!
(11_FOSEGED\PMASTDET.DPR)

11.5.2 Feladat: Névböngésző kezdőbetű alapján

Készítsük el a következő alkalmazást:

Alkalmazásunk egyetlen űrlapján legyen egy fülsor az ábécé betűivel és egy rács, melynek mindig a kiválasztott kezdőbetűs vevőket kell tartalmaznia. A vevők adatai a *DBDEMOS* álnév *Customer* táblájából származzanak (a Delphi minta-adatbázisából).



11.12. ábra. Névböngésző kezdőbetű alapján

Megoldás (🖫 11_NEVBONGESZO\PBONGESZO.DPR)

A rács tartalmát szolgáltató lekérdezésnek a következőképpen kellene kinéznie:

```
{Ha az 'A' fül van kiválasztva}
SELECT * FROM Customer
WHERE Company LIKE 'A%'

{Ha az 'B' fül van kiválasztva}
SELECT * FROM Customer
WHERE Company LIKE 'B%'

...
```

Ehelyett, a lekérdezés változó részét egy paraméterrel fogjuk helyettesíteni:

```
SELECT * FROM Customer
WHERE Company LIKE :P + '%'
```

Hozzunk létre egy adatmodult (*DM*), helyezzünk el rajta egy *qCust:TQuery* és egy *dsrqCust:TDataSource* komponenst. Írjuk be a fenti utasítást a lekérdezés komponens *SQL* jellemzőjébe. Mivel a *qCust* egy paraméteres lekérdezést tartalmaz, és mivel ezt a lekérdezést sokszor fogjuk lefuttatni, ajánlatos meghívni a *Prepare* metódusát. Ezt az adatmodul *OnCreate* eseményébe írjuk:

```
procedure TDM.DMCreate(Sender: TObject);
begin
  qCust.Prepare;
end;
```

Alkalmazásunk űrlapján helyezzük el a *Fulsor:TTabControl* és a *Racs:TDBGrid* komponenseket. A rácsot irányítsuk az adatmodulon levő adatforrásra.

A fülsort programból fogjuk feltölteni az ábécé betűivel. Amikor a felhasználó átkattint egy másik fülre, be kell zárnunk a lekérdezést, át kell állítanunk paraméterét az új fül feliratára, majd újra meg kell nyitnunk. Ezt a fülsor *OnChange* eseményébe írjuk.

```
procedure TfrmBongeszo.FormCreate(Sender: TObject);
Var C:Char;
begin
  Fulsor.Tabs.Clear;
  For C:= 'A' To 'Z' Do
    Fulsor.Tabs.Add(C);
  {kezdetben az 'A' legyen kijelölve a fülsoron is és a rácsban is}
  Fulsor.TabIndex:=0;
  FulsorChange(self);
end;

procedure TfrmBongeszo.FulsorChange(Sender: TObject);
begin
  With DM.qCust Do
    begin
      Close;
      Params[0].AsString:= Fulsor.Tabs[Fulsor.TabIndex];
      Open;
    end;
end;
```

Itt könnyen általános védelmi hibával találkozhatunk. Miért?
A probléma az, hogy az űrlap létrehozásakor (*FormCreate*) meghívjuk a *FulsorChange* metódust, abban pedig az adatmodulra hivatkozunk. Ha az adatmodul csak az űrlap megszületése után jön létre, akkor máris megvan a hiba. Orvoslása: hívjuk meg a *Project/Options* menüpontot, jelöljük ki ennek *Forms* lapját, és az automatikus létrehozású űrlapok között változtassuk meg (vonszolással) a sorrendet: első legyen az adatmodul és második az űrlap.

Hogyan lehetne másképp megoldani a feladatot?

Oldja meg ugyanezt a feladatot szűrt (*Filter*) táblakomponenssel!

12. Feladat: A könyvnyilvántartó folytatása

A 10. fejezetben kitűzött feladat nagyobbik részét már ott megoldottuk, csak a lekérdezések és a jelentések maradtak hátra. Ebben a fejezetben kiegészítjük a szükséges lekérdezésekkel:

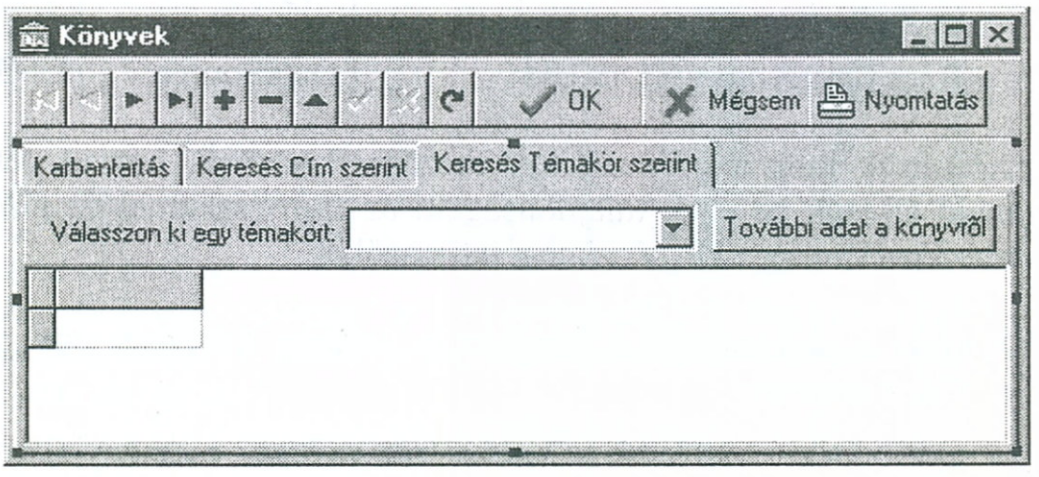
- A könyvek témakör szerinti kereséséhez felépítünk egy paraméteres lekérdezést.
- A *Könyvek* űrlapon megjelenítjük az aktuális könyv szerzőinek számát ugyancsak egy paraméteres lekérdezés segítségével.

Megoldás (🖫 12_KONYVTARFOLYTATAS\PKONYVTAR.DPR)

12.1 Könyvek keresése témakör szerint

Lépjünk az *frmKonyv* harmadik, még üres oldalára. Itt a felhasználó választhasson ki egy kombinált listából egy témakört. A mi feladatunk az, hogy leválogassuk, majd megjelenítsük a megfelelő témájú könyveket. Egy gombbal biztosítanunk kell a kikeresett könyv további adatainak megtekintését is.

A témaköröket tartalmazó kombinált lista *DBLookupComboBox* típusú lesz, hiszen csak ennél a komponensnél állítható be a lebomló lista forrása. Nálunk ez a *tblTema* táblakomponens lesz. Ugyanakkor, ha a felhasználó kiválaszt a listából egy témakört, az értékének nem szabad egyik táblába sem beíródnia. A kiválasztott témakör csak szűrési szempont. A kombinált listát tehát semmihez sem kötjük (a *DataSource* és *DataField* jellemzői üresen maradnak).



12.1. ábra. A könyvek űrlap témakör szerinti keresés oldala

A rács egy paraméteres lekérdezés eredményét fogja tartalmazni, a paraméter értékét pedig a kombinált lista szolgáltatja. Tervezzük tehát meg az adatmodulon a lekérdezést, majd az űrlap témakör szerinti keresés lapját is.
A lekérdezésben össze kell gyűjtenünk a könyvről szóló információkat: *ISBN*, *Cím*, *Kiadó*, *Téma*, *Példányszám*, *Leírás*. Ezeket az információkat összesen három táblából szedjük össze a következők szerint:

```
SELECT ISBN, Cim, Kiado, Tema, Peldanyszam, Leiras
FROM Konyv, Tema, Kiado
WHERE
(Konyv.KiadoAz=Kiado.KiadoAz)
AND (Konyv.TemaAz = Tema.TemaAz)
AND (TemaAz = :P)
```

Az adatmodulra és űrlapra elhelyezendő komponensek beállításait a következő táblázat mutatja:

Komponens	Jellemző	Érték
A DMKonyvtar adatmodulra:		
qTemaKonyv:TQuery	DatabaseName	DBKonyvtar
	SQL	*{a fenti SQL utasítás}*
	Params. DataType	Integer
dsrqTemaKonyv:TDataSource	DataSet	qTemaKonyv
Az frmKonyv űrlap 'Keresés témakör szerint' oldalára:		
dblcTemaKeres:TDBLookupComboBox	DataSource	– *{nem ír sehova}*
	DataField	–
	ListSource	DMKonyvtar.tblTema
	ListField	Tema
	KeyField	TemaAz
btnTovAdat:TButton	Caption	'További adat a könyvről'
RacsTemaKonyv:TDBGrid	DataSource	DMKonyvtar.dsrqTemaKonyv

Jelenítsük meg a *qTemaKonyv* mezőszerkesztőjét, hozzuk létre a perzisztens mezőket, majd állítsuk be ezek tulajdonságait: az *ISBN* számnál az *EditMask*, *DisplayWidth*, más mezőknél pedig a *DisplayLabel* jellemzőket.

Amikor a felhasználó a kombinált listában kiválaszt egy témakört, bekövetkezik az *OnClick* esemény.

Az *TDBLookupComboBox.OnClick* eseményjellemzője megtévesztő nevet visel, tudniillik nem csak az egér kattintására hívódik meg, hanem akkor is, ha a billentyűzetről váltunk listaelemet. Találóbb lett volna az *OnChange* elnevezés (Delphi 1-ben volt is ilyen).

A kombinált lista *OnClick* eseményében meg kell vizsgálnunk, hogy az új témakör megegyezik-e vagy nem az előzővel. Csak akkor van értelme újrafuttatni a lekérdezést, ha a felhasználó egy új témakört állított be. (Ezzel időt takarítunk meg, és elkerüljük a kellemetlen villogást, amit a rács frissítése okozna.)

```
procedure TfrmKonyv.dblcTemakeresClick(Sender: TObject);
begin
  inherited;
  With DMKonyvtar Do
    begin
      If tblTemaTemaAz.AsInteger <> qTemaKonyv.Params[0].AsInteger
        Then
          With qTemaKonyv Do
            begin
              Close;
              Params[0].AsInteger:=tblTemaTemaAz.AsInteger;
              Open;
            end;
    end;
end;
```

Ha most elindítjuk az alkalmazást, akkor már működik a leválogatás. Egy „szépséghibát" azonban biztosan felfedeztünk: kezdetben még nincs egyetlen témakör sem kiválasztva, így a rács is üres. De hogyan lehet egy *DBLookupComboBox* tartalmát beállítani? Ez általában annak az adatforrásnak aktuális rekordbeli mezőértékét mutatja, melyhez kötöttük (*DataSource, DataField*); most azonban egy „kötetlen" kombinált listánk van, hiszen a kiválasztott témakört nem szeretnénk egyetlen adathalmazba sem beírni.

A megoldás a kombinált lista *KeyValue* jellemzőjében rejlik. Ebbe a *KeyField* jellemző (nálunk *TemaAz* mező) megfelelő értékét kell beállítanunk. A következő kódrészlet ezt mutatja: (csak a félkövéren szedett utasításokat írjuk most be, a többi már megvolt.)

```
procedure TfrmKonyv.FormCreate(Sender: TObject);
begin
   inherited;
   KeresesMezo:='Cim';
   {beállítjuk a kombinált lista kezdeti értékeként az éppen
   aktuális témakört, majd
   a dblcTemaKeresClick hívásával a lekérdezést is lefuttatjuk}
   dblcTemaKeres.KeyValue:= DMKonyvtar.tblTemaTemaAz.AsString;
   dblcTemaKeresClick(self);
end;
```

Már csak a rácsban kiválasztott könyv további adatait kell megjelenítenünk a *btnTovAdat* gomb *OnClick* eseményében. Ehhez meg fogjuk jeleníteni az *frmKonyv* űrlap *Karbantartás* oldalát. Igen ám, de előtte még biztosítanunk kell azt, hogy ezen az oldalon a témakör szerinti keresés eredményeként kiválasztott könyv adatait láthassuk.

Amikor a felhasználó a rácsban egy rekordra kattint, akkor tulajdonképpen a rács alapjául szolgáló lekérdezésben a mutatót erre a rekordra pozícionálja; tehát a *qTemaKeres* aktuális rekordjában található *ISBN* szám alapján a *tblKonyv*-ben rá kell keresnünk a megfelelő könyvre.

```
procedure TfrmKonyv.btnTovAdatClick(Sender: TObject);
begin
   inherited;
   DMKonyvtar.tblKonyv.
        Locate('ISBN',DMKonyvtar.qTemaKonyvISBN.AsString, []);
   Oldalak.ActivePage:= Oldalak.Pages[0];
end;
```

Tesztelje le az alkalmazást!

12.2 Egy könyv szerzőinek megszámlálása

Az *frmKonyv Karbantartás* oldalán a könyvek adatai mellett jelenítsük meg az aktuális könyv szerzőinek számát is. Ezt kiszámolhatnánk a szerzőket megjelenítő rács sorainak számából (programban ez a *tblKonyvSzerzoi* tábla rekordjainak számát jelentené), vagy kiszámolhatjuk egy lekérdezéssel is. Mindkettő tanulságos, így azt javaslom, oldjuk meg mindkét módszerrel.

Első megoldás:

Egy tábla rekordjainak számát a *RecordCount* jellemzője tartalmazza. Ezt az értéket fogjuk egy címkében *(lSzerzokSzama)* megjeleníteni az *frmKonyv* űrlapon. A címke az aktuális könyv szerzőinek számát mutatja, tehát egy új szerző felvételekor a címke tartalmának is változnia kellene. Igen ám, de milyen eseményre építsük be ezt a frissítést? Ha felveszünk egy új szerzőt, vagy kitörlünk egyet, vagy egyszerűen átlépünk egy másik könyvre stb., ezek mind olyan tevékenységek, melyek befolyásolják a szerzők számát. Tulajdonképpen ilyenkor a *tblKonyvSzerzoi* táblában történnek változások. A ráirányított adatforrás *(dsrKonyvSzerzoi) OnDataChange* eseménye minden adatváltozáskor bekövetkezik. Ez magába foglalja az előbb felsorolt eseteket is. Erre építjük tehát a címke frissítését.

```
procedure TDMKonyvtar.dsrKonyvSzerzoiDataChange(Sender: TObject;
                                                Field: TField);
begin
   {Az adatmodul hamarabb jön létre, mint az frmKonyv űrlap.
   Ha nem teszteljük le az űrlap létezését, akkor garantáltan
   „összefutunk" egy AccessViolation hibával}
   If frmKonyv<>Nil Then
      frmKonyv.LSzerzokSzama.Caption:=
                     IntToStr(tblKonyvSzerzoi.RecordCount);
end;
```

Metódusunk akkor is lefut, ha csak rekordon belül módosítunk, például amikor egy szerzőt kicserélünk egy másikra. Ez az eset nincs hatással a szerzők számára, ekkor kihagyhatnánk a címke frissítését, de sajnos, nem tehetjük.

Ha most lefuttatjuk az alkalmazást, akkor ugyan nincs *AccessViolation* hiba, de kezdetben a címke is üres. Ez azért van, mert amikor legelőször lefut a *TDMKonyvtar.dsrKonyvSzerzoiDataChange* metódus, akkor az *frmKonyv* még nem létezik (csak az adatmodul létrejötte után kerül rá a sor), így hát a címke tartalmának beállítása is elmarad.
Ezt az *frmKonyv.OnCreate* eseményében pótolnunk kell:

```
procedure TfrmKonyv.FormCreate(Sender: TObject);
begin
   inherited;
   ...
   LSzerzokSzama.Caption:=
          IntToStr(DMKonyvtar.tblKonyvSzerzoi.RecordCount);
end;
```

Második megoldás:

Számoljuk össze a könyv szerzőit egy lekérdezés segítségével. Lekérdezésünk paraméteres lesz, hiszen mindig az aktuális könyv szerzőit kell tartalmaznia.

```
{DMKonyvtar.qSzerzokSzama.SQL}
SELECT Count(*) FROM Szerzo
WHERE ISBN = :ISBN
```

A paraméter a *tblKonyv* aktuális rekordjának *ISBN* értékéből származik. Állítsuk be a lekérdezés *(qSzerzokSzama) DataSource* jellemzőjét a *dsrKonyv* adatforrásra. Ezek után már csak a paraméter típusát kell beállítanunk (*String*-re), és máris megnyithatjuk a lekérdezést. A könyvek űrlapon egy *DBText* komponensben jelenítsük meg a *'Count(*)'* mező értékét, majd teszteljük le alkalmazásunkat.

Azt vesszük észre, hogy a szerzők számlálása egyelőre még csak félig működik, hiszen a könyv változásakor frissül, de a szerzők felvételénél és törlésénél nem. A *btnUjSzerzo* és a *btnSzerzoTorlese* gombokra a lekérdezést újra le kell futtatnunk.

```
procedure TfrmKonyv.btnUjSzerzoClick(Sender: TObject);
begin
   inherited;
   ...
   DMKonyvtar.qSzerzokSzama.Close;
   DMKonyvtar.qSzerzokSzama.Open;
end;
```

```
procedure TfrmKonyv.btnSzerzoTorleseClick(Sender: TObject);
begin
   inherited;
   DMKonyvtar.tblKonyvSzerzoi.Delete;
   DMKonyvtar.qSzerzokSzama.Close;
   DMKonyvtar.qSzerzokSzama.Open;
end;
```

Miért futtatjuk le még egyszer a lekérdezést? Miért nem növeljük eggyel a szerzők számát az új író felvételénél, és miért nem csökkentjük a törlésnél?

Ennek két oka van:

- Az első az, hogy a szerzők száma a lekérdezés *'Count(*)'* mezőjéből származik, így ennek értékére nem is tudunk igazán hatni. A lekérdezésben ez a mező nem szerkeszthető.
- A második az, hogy még ha tudnánk is egyel növelni vagy csökkenteni a szerzők számát, akkor sem biztos, hogy ezzel reális eredményt kapnánk. Amikor egy új szerzőt felveszünk, könnyen előfordulhat, hogy egy már létezőt próbálunk másodszor beállítani, és ezzel a *tblKonyvSzerzoi* táblában kulcsismétlést idézünk elő. Ezt a *btnUjSzerzoClick* metódusban nem tudjuk lekérdezni (majd csak a *tblKonyvSzerzoi.OnPostError* eseményében derül ki, lásd 10. fejezet), így „tudatlanul" növelni nem szabad a szerzők számát.

Jelenítse meg az *Írók* űrlapon az aktuális író könyveinek számát!

Tipp: ebben a feladatban ugyancsak egy paraméteres lekérdezést hoznánk létre, a paramétert pedig a *tblIro IroAz* mezőjéből olvasnánk ki. Igen ám, de ez a mező számláló típusú, általában pedig a lekérdezések paramétere nem lehet ilyen típusú.
Egy furfangos megoldás az lenne, hogy állítsuk a lekérdezés *DataSource* jellemzőjét a *dsrIroKonyvei* adatforrásra. Ebben az *IroAz* értéke mindig követi az írók táblában levő író azonosítóját, és ugyanakkor ennek típusa már nem *autoincrement*.

Tud erre esetleg más megoldást?
Tipp: olvassa el figyelmesen a 8. fejezetben a 8.5.2. pontot.

13. Jelentések

A Delphi 32 bites verzióiban jelentések készítéséhez a beépített *QuickReport* komponenscsaládot (*QReport* paletta) használhatjuk. Ez egy ún. *„third party"* komponenscsomag, mely a jelentések vizuális tervezését teszi lehetővé.

A 16 bites verzióban még nem álltak rendelkezésünkre beépített jelentés-komponensek, így egy segédprogrammal (a ReportSmithel) lehetett listákat készíteni. Kezdetben mindenki ezt használta, aztán később, a lassúsága miatt, átálltak inkább a kódból létrehozott listákra, vagy a már akkor megjelent *QuickReport* komponensek használatára. Könyvünkben emiatt nem foglalkozunk a ReportSmith programmal. A 16 bites Delphivel rendelkező Olvasóknak azt javaslom, hogy töltsék le a *www.qusoft.com* címről a *QuickReport* komponenscsalád 16 bites változatát. Ez egy kicsit más, szegényesebb, mint a Delphi 3-ba beépített verzió, de bízom abban, hogy az itt bemutatottak alapján el tudnak majd igazodni abban is.

Ebben a fejezetben tehát a jelentések *QuickReport* komponenscsalád segítségével történő készítésének menetét ismertetjük.

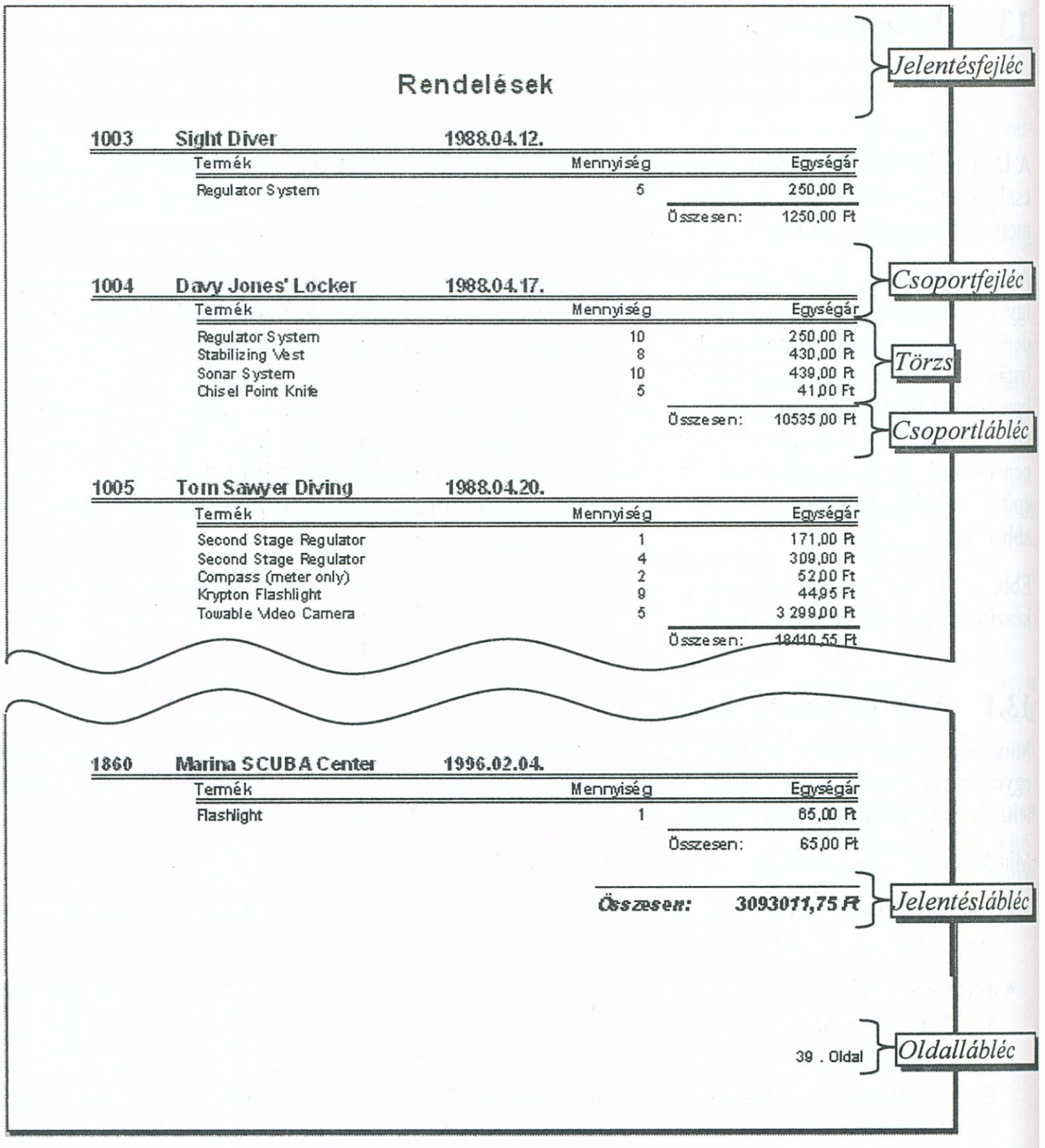
13.1 A jelentések felépítése

Minden jelentés egy adathalmaz tartalmát listázza ki. Példánkban (13.1. ábra) minden egyes megrendelésről megjelenítjük az azonosítóját, vevőjét, vásárlás dátumát, és a tételeit. Készítettünk tehát egy lekérdezést ezen információk összegyűjtésére.

Minden jelentés szakaszokból áll. A szakaszokra a megjelenési helyük jellemző:

- **Jelentésfejléc**: egyszer jelenik meg a jelentés legelső oldalán. Ebben szoktuk feltüntetni a jelentés címét.
- **Jelentéslábléc**: a jelentés utolsó oldalán jelenik csak meg, benne végösszegzéseket szoktunk megjeleníteni.
- Ha csoportosításokat is végzünk jelentésünkben, akkor a **csoport**nak lehet egy **fejléc**e és egy **lábléc**e. Példánkban a megrendelés azonosítója szerint csoportosítottunk. Egy csoportban az adott megrendelés tételeit tüntettük fel. Csoportonként egyszer jelenjen meg a megrendelés azonosítója, vevője és dátuma, ezért ezeket a csoportfejlécében helyeztük el. Ugyancsak a fejlécbe került a törzsben következő információk oszlopfejléce is (*Termék, Mennyiség, Egységár*). A csoport láblécében most egy összegzést jelenítettünk meg.

- **Jelentéstörzs**: az ide helyezett információk a forrás-adathalmaz minden egyes rekordjára megismétlődnek. Példánkban itt tüntettük fel a termék nevét, mennyiségét és egységárát, így ezek minden egyes megrendeléstétel esetében megjelentek a konkrét értékeikkel.



13.1. ábra. Egy jelentés felépítése

- Ezeken kívül a jelentésben létrehozhatunk még **oldalfejléc és -lábléc** szakaszokat is. Itt leginkább általános információkat szoktunk megjeleníteni, mint például az oldalszámot, a jelentést készítő cég emblémáját, a jelentés címét stb.
- Jelentésünk tartalmazhat még **oszlopfejléc** szakaszt is, ez minden oldalon megjelenik a lista oszlopainak fejlécével.

13.2 A QuickReport komponenscsalád

Tekintsük át a fontosabb *QuickReport* komponenseket:

Szerkezeti komponensek:

TQuickRep — *A **jelentést** képviselő komponens*

TQRSubDetail — *A **beágyazott jelentés** komponense. Bővebben lásd a 13.4.3.2 pontban.*

TQRBand — ***Szakasz** komponens. A BandType jellemzője tartalmazza a szakasz típusát.*

TQRGroup — ***Csoportosítás**kor használjuk. Ez egyben a csoport fejlécének szakasza is. Meg kell adnunk a csoportosítási szempontot. Ha a csoporthoz lábléc is tartozik, akkor azt is itt kell beállítanunk.*

Megjelenítési komponensek:

TQRLabel — ***Címke**: benne egy fix szöveget jeleníthetünk meg (pl. jelentés címe, oszlopfejlécek stb.)*

TQRDBText — ***Szerkesztődoboz**: egy adathalmazból származó értékek megjelenítésére használjuk.*

TQRExpr — ***Kifejezések** (pl. összegzések) kiszámolására és megjelenítésére használjuk.*

TQRSysData — ***Rendszeradatok** (pl. dátum, idő, oldalszám) megjelenítésére használjuk.*

TQRDBImage — *Tárolt **képeket** jeleníthetünk vele meg.*

TQRChart — ***Diagramok** készítésére használjuk.*

A következőkben példákon keresztül fogunk megismerkedni az itt felsorolt komponensekkel. Előbb azonban tekítsük át egy jelentés készítésének lépéseit.

13.3 A jelentések készítésének lépései

- **Az adatok összegyűjtése:** ha a kilistázandó adatok egyetlen fizikai táblából származnak, akkor a jelentés alapját egy erre irányított *TTable* komponens képezi. Ha már több táblából kell az adatokat „összevadásznunk", vagy ha egy nagy táblából csak néhány mezőt szeretnénk kilistázni, akkor *TQuery* komponenst használjunk. A jelentés

alapjául akár egy *TStoredProc* komponens is állhat, ennek használatát lásd a következő fejezetben. A lényeg tehát az, hogy összeszedjük a jelentés adatait.

- **A jelentés megtervezése:** egy **új űrlapon** össze kell állítanunk a jelentéstervet:
  - Elhelyezünk űrlapunkon egy *QuickRep* komponenst. Adatforrását (*DataSet* jellemzőjét) beállítjuk az előzőleg elkészített adathalmaz-komponensre. Itt végezhetjük el a különböző oldal- és nyomtató-beállításokat is (*Page, Printer Settings*).
  - A jelentés területére elhelyezzük a szükséges szakaszokat. Mindegyiküknél beállítjuk típusát (*BandType*) és nevét (*Name*). Egyeseknél még további adatokat is meg kell adnunk (például a *QRGroup* komponensnél specifikálnunk kell a csoportosítási szempontot is).
  - Összeállítjuk a szakaszok tartalmát a különböző *QRLabel, QRText, QRExpr, QRSysData, QRDBImage* stb. komponensek segítségével.

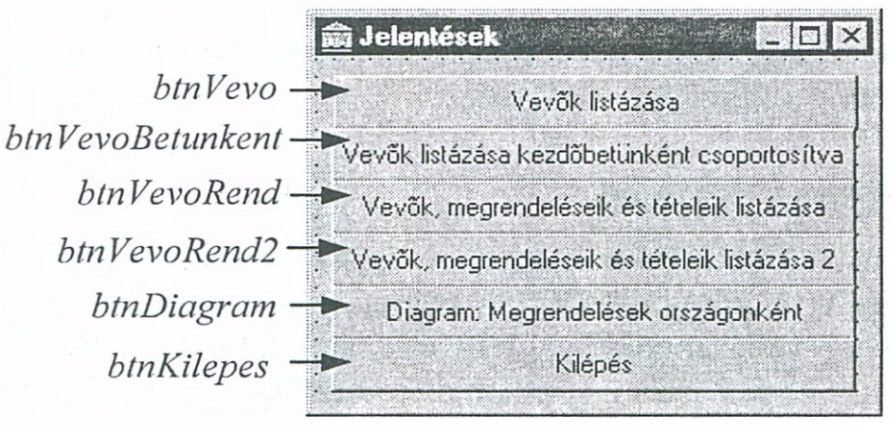
  A készülőben levő jelentést bármikor megjeleníthetjük a jelentéskomponens gyorsmenüjéből a *Preview* parancs segítségével.
- **A jelentés megjelenítése programkódból.** Ez a jelentéskomponens *Preview* vagy *Print* metódusának meghívásával történik.

13.4 Jelentések példákon keresztül

A jelentések készítésének bemutatására készítsünk egy alkalmazást öt jelentéssel. Mindvégig a már ismert *DBDEMOS* álnév alatti *Customer*, *Orders*, *Items* és *Parts* táblákat fogjuk használni. Az öt jelentés a következő lesz:

- Vevők listázása azonosítójuk sorrendjében
- Vevők listázása a nevük kezdőbetűje szerint csoportosítva
- Vevők, megrendeléseik és azok tételeinek listázása 1
- Vevők, megrendeléseik és azok tételeinek listázása 2 (feltüntetjük egy adott vevő megrendeléseinek számát is)
- Diagram: megrendelések országonként

Megoldás (💾 13_JELENTESEK\PJELENTES.DPR)



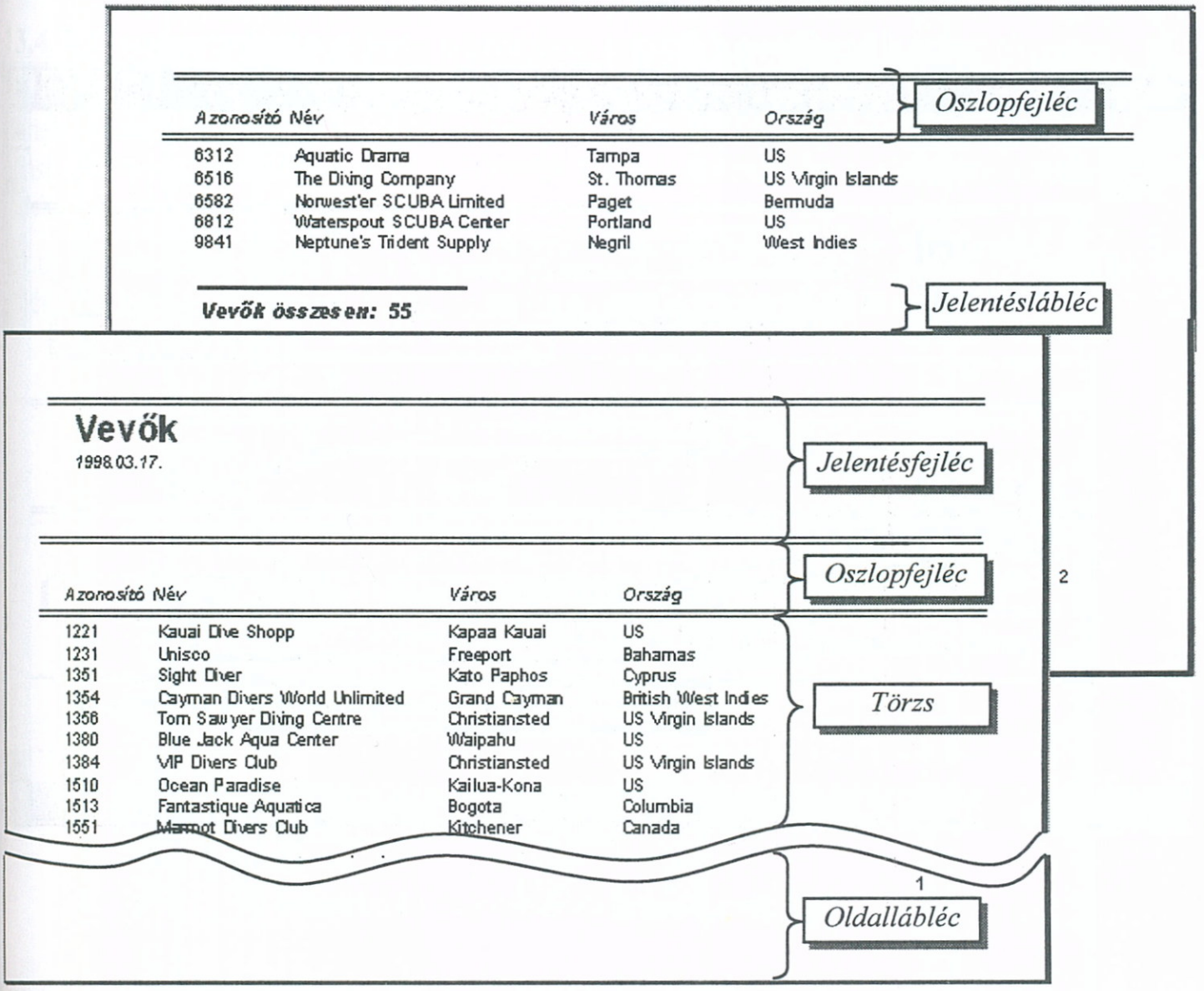
Készítsünk egy alkalmazást a 13.2. ábrán látható főűrlappal. A gombok segítségével fogjuk a négy jelentést megjeleníteni nyomtatási előképben.

13.2. ábra. Minden gombra egy-egy jelentés

13.4.1 Egyszerű jelentés létrehozása: vevők listázása

Hozzunk létre egy új űrlapot (*frmqrVevo*). Ezen fogjuk a jelentést megtervezni. Helyezzünk el rajta egy táblakomponenst (*tblVevo*), és irányítsuk a *DBDEMOS/Customer* tábla felé. A vevőket azonosítójuk sorrendjében kell kilistáznunk, tehát a táblában megtarthatjuk az alapértelmezett indexet.

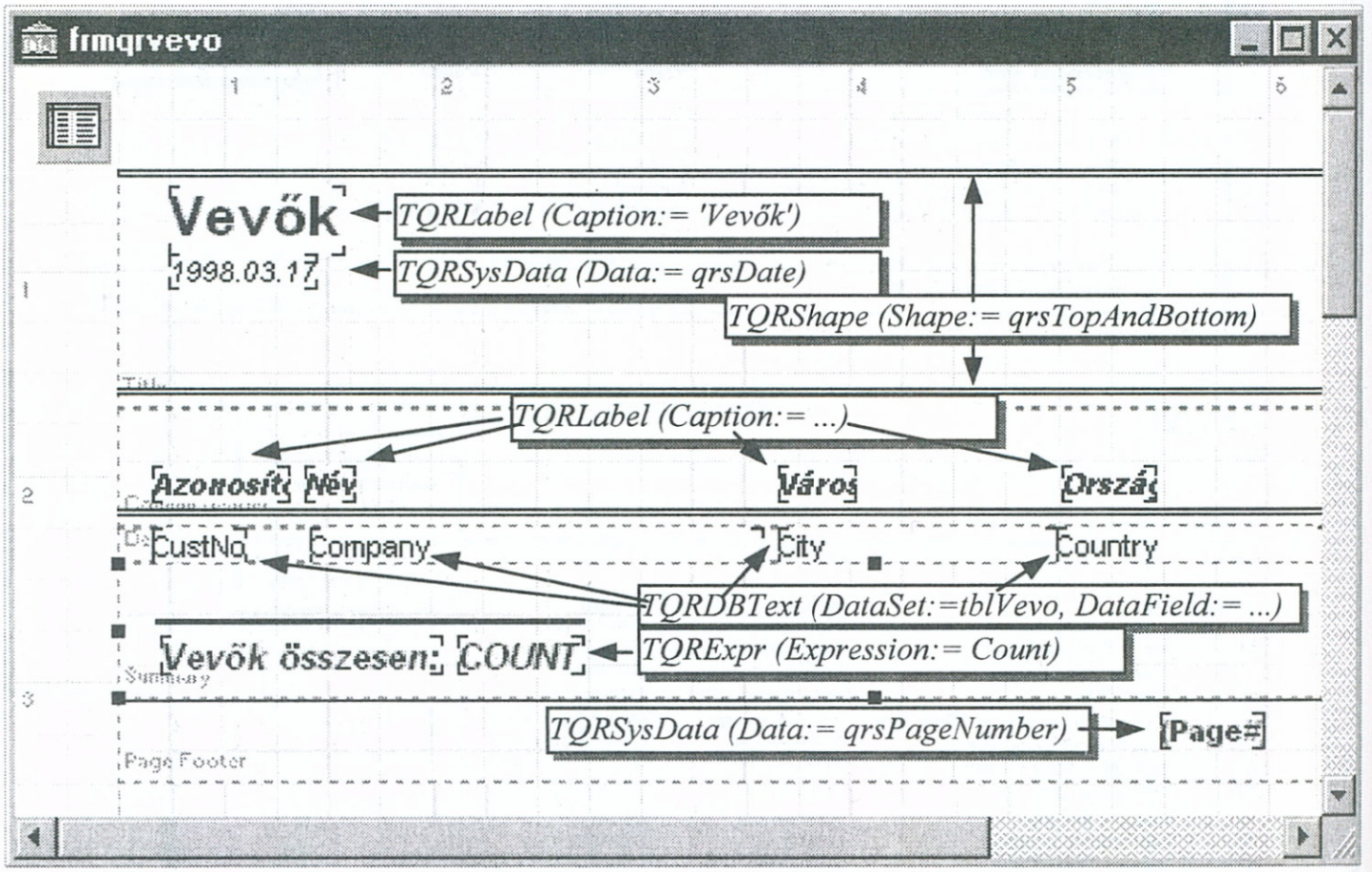
Helyezzünk el űrlapunkon egy *QuickRep* komponenst. Nevezzük el *qrVevo*-nek, majd állítsuk be *DataSet* jellemzőjét *tblVevo*-re.



13.3. ábra. Vevők listája. Összesen öt szakaszból áll.

Helyezzük el a jelentés területére (egymás alá) az öt szakaszt öt *QRBand* komponens formájában. *BandType* jellemzőjüket állítsuk sorban az *rbTitle* (jelentésfejléc), *rbColumnHeader* (oszlopfejléc), *rbDetail* (törzs), *rbSummary* (jelentéslábléc) és *rbPageFooter* (oldallábléc) értékekre.

Helyezzük el a szakaszokban a megjelenítendő információkat a 13.4. ábra alapján. Zárójelben a beállítandó jellemzőket tüntettük fel. A statikus szövegeket *TQRLabel* komponensek segítségével jeleníthetjük meg: feliratukat (*Caption*) állítsuk az ábra szerintire. Az adatforrásból származó információknak egy-egy *TQRDBText* komponenst kell elhelyeznünk a törzsszakaszban. *DataSet* jellemzőjüket állítsuk a *tblVevo*-re, a *DataField*-et pedig a megfelelő mezőre. A nyomtatás dátumát és az oldalszámot *TQRSysData* komponensekkel jelenítjük meg. A jelentés láblécében a vevők számát egy *TQRExpr* komponens segítségével számoljuk ki, melynek *Expression* jellemzőjébe a *Count* függvényt írjuk be. Ezzel a törzsben szereplő rekordok számát fogjuk összeszámolni.



13.4. ábra. Jelentésünk tervező nézetben

Ha végeztünk a beállításokkal, akkor teszteljük le a jelentést a gyorsmenü *Preview* parancsával (ez a gyorsmenü akkor jelenik meg, ha valahol a margó felületén kattintunk az egér másodlagos gombjával).

Jelenítsük meg a jelentést programból is. A főűrlap *btnVevo* gombjára építsük be a következő metódust (A jelentés forrását képező táblát vagy lekérdezést ekkor nyitjuk meg, majd a nyomtatás után bezárjuk. Így mindig friss adatokat listázunk ki, és az erőforrásokkal is okosan gazdálkodunk):

```
procedure TfrmMain.btnVevoClick(Sender: TObject);
begin
  With frmqrVevo Do
    begin
      tblVevo.Open;
      qrVevo.Preview;
      tblVevo.Close;
    end;
end;
```

13.4.2 Csoportváltásos lista készítése: vevők kezdőbetűk szerint

Még mindig a CUSTOMER.DB tábla tartalmát listázzuk ki, de most már névsorrendben. Ennek érdekében az új jelentés űrlapjára (*frmqrBetunkent*) helyezzünk el egy táblakomponenst (*tblVevo*), irányítsuk a CUSTOMER.DB-re, állítsuk be a *Company* mező szerinti indexet, majd nyissuk meg.

Az új jelentéskomponens neve legyen *qrVevoBetunkent*. Ne felejtsük el *DataSet* jellemzőjét beállítani a *tblVevo*-re. Szerkezetét a következő ábrán tanulmányozzuk:

Vevők

1998.03.17.

A

Azonosító	Név	Város	Ország
1645	Action Club	Sarasota	US
3158	Action Diver Supply	St. Thomas	US Virgin Islands
1984	Adventure Undersea	Belize City	Belize
3053	American SCUBA Supply	Lomita	US
6312	Aquatic Drama	Tampa	US

B

Azonosító	Név	Város	Ország
3984	Blue Glass Happiness	Santa Monica	US
1380	Blue Jack Aqua Center	Waipahu	US
1563	Blue Sports	Giribaldi	US
2118	Blue Sports Club	Largo	US

C

Azonosító	Név	Város	Ország
3054	Catamaran Dive Club	Catalina Island	US
1354	Cayman Divers World Unlimited	Grand Cayman	British West Indies
5151	Central Underwater Supplies	Johannesburg	Republic So. Africa

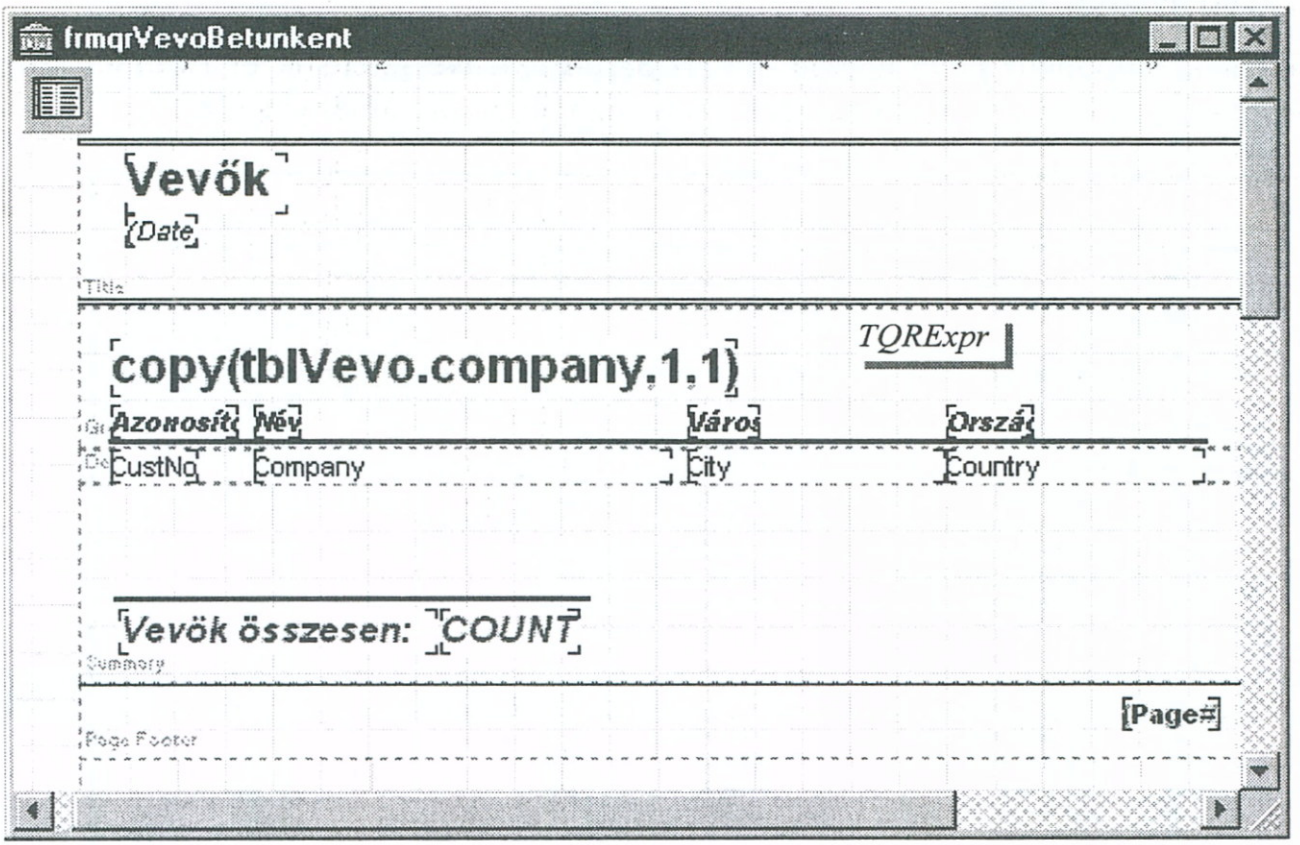
13.5. ábra. Vevők csoportváltásos listája

A jelentés szakaszai a következők: jelentésfejléc (benne a jelentés címe), csoportfejléc (a csoport betűje és az oszlopfejlécek), törzs (a vevők adatai), oldallábléc (az oldalszám) és a jelentéslábléc (a vevők száma). Minden szakasznak – a csoportfejléc kivételével – megfelel egy-egy *TQRBand* komponens. A csoport fejlécét *TQRGroup*-pal jelenítjük meg.

A *TQRGroup* komponens *Expression* jellemzőjébe a csoportosítás szempontját kell beírnunk. A mi esetünkben ez nem más, mint a vevő nevének (*Company*) első betűje, tehát: *Expression := Copy (tblVevo.Company, 1, 1)*. A csoportosítási szempontot képező kifejezést vagy begépeljük közvetlenül az objektum-felügyelőben, vagy mindezt a kifejezésszerkesztő segítségével tesszük. Ha az így definiált csoportnak lábléce is lenne, akkor ennek nevét a *TQRGroup* komponens *FooterBand* jellemzőjébe kellene írnunk. Példánkban a kezdőbetű csoportokhoz nem tartozik lábléc, ezért a csoportfejléc komponens *FooterBand* jellemzőjét üresen hagyjuk.

A csoport betűjét egy *TQRExpr* komponenssel jelenítjük meg; ennek *Expression* jellemzőjébe ugyancsak a *Copy (tblVevo.Company, 1, 1)* képletet írjuk be.

Helyezze el a jelentés különböző szakaszain a megfelelő komponenseket a 13.6. ábra alapján!

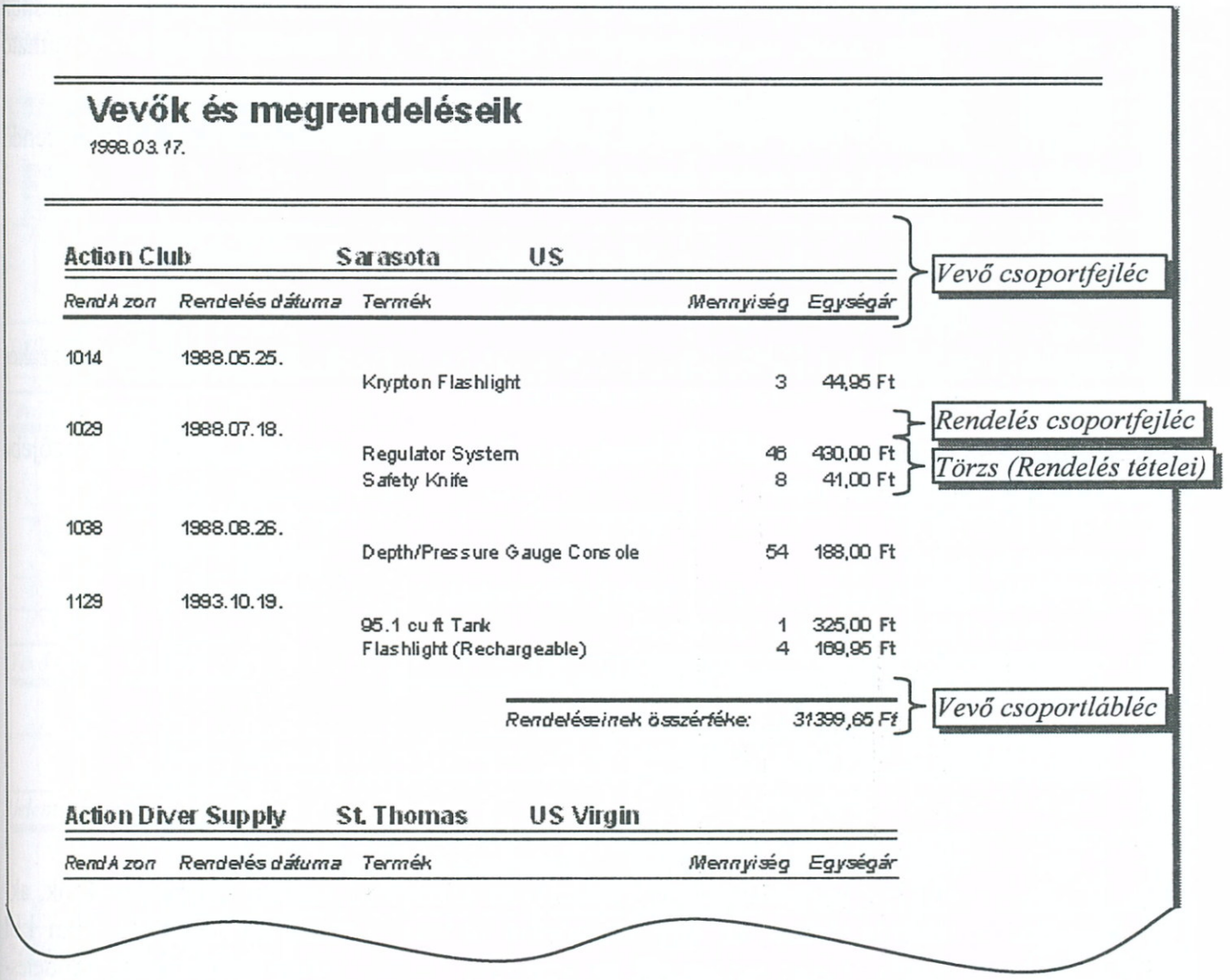


13.6. ábra. Jelentésünk tervező nézetben

Fejlessze tovább a jelentést úgy, hogy a csoportok végén tüntesse fel a csoportban megjelenő vevők számát. Tipp: ugyancsak *TQRExpr* komponenssel számoljuk ezt is ki, a *Count* függvénnyel, akárcsak a jelentés láblécében levőt. A különbség a két kifejezés között az, hogy a csoportláblécben levő összeget le kell nulláznunk minden csoport végén, a kinyomtatása után. Ennek érdekében a csoportláblécben levő *TQRExpr* komponens *ResetAfterPrint* jellemzőjét Igazra kell állítanunk.

13.4.3 Kétszintű csoportváltásos lista: vevők, megrendeléseik és tételeik

Egy vevőnek több megrendelése lehet, egy rendelésben pedig több tétel is szerepelhet. Ezen adatok listázásakor csoportosítanunk kell vevő, és azon belül rendelés szerint. Emiatt jelentésünk kétszintű csoportváltást fog tartalmazni.



13.7. ábra. Kétszintű csoportváltásos lista

A *QuickReport* komponensekkel többszintű csoportváltásos listát kétféleképpen is készíthetünk: használhatunk több *TQRGroup* komponenst, vagy használhatjuk a beágyazott jelentéskomponenst – a *TQRSubDetail*-t. (Accesses terminológiában ezt segédjelentésnek nevezzük). Elemezzük e két megoldást:

Ha **az első megoldás** mellet döntünk (13.7. ábra), akkor az egész jelentésben csak egy törzsszakaszunk lesz (példánkban ez a rendelés tételeit tartalmazza). De miért érdekes ez? Amint már észrevettük, a kifejezésekben a *COUNT*, *SUM*... aggregáló függvények a törzsszakasz soraira vonatkoznak. Ebből kifolyólag egy jelentésben csak olyan összegzéseket tudunk kiszámolni, melyek a törzsszakasz adatain alapulnak.

Az első megoldásban tehát egy vevő rendeléseinek összértéke kiszámolható a törzsrekordokban szereplő *Mennyiség* és *Egységár* szorzatának összegzésével. Egy vevő megrendeléseinek számát viszont már nem tudnánk kiszámolni, mivel ez az információ nem található meg a törzsszakaszban (ez a *Rendelés* csoportfejlécek darabszáma lenne).

A második megoldásban tulajdonképpen két törzsszakaszunk lenne: egy a főjelentésnek és egy második a beágyazott, segédjelentésnek. Így ezzel a módszerrel már kiszámítható lesz a vevő rendeléseinek száma, valamint ezek összértéke is.

A jelentés megtervezése előtt tehát ajánlatos aprólékosan kigondolni a megjelenítendő információkat. Készítsük el mindkét megoldással a jelentést.

13.4.3.1 Első megoldás: többszintű csoportosítás (két TQRGroup komponenssel)

Első megoldásban tehát két *TQRGroup* komponenst használunk, így egyetlen törzsszakaszunk lesz. Gyűjtsük előbb össze a kinyomtatandó adatokat!

Helyezzünk el űrlapunkon *(frmqrVevoRend)* egy *TQuery* komponenst, *SQL* jellemzőjébe pedig írjuk a következő utasítást:

```
{qVevoRendTetelek.SQL}
SELECT  Company, Addr1, City, Country, OrderNo, Saledate,
        Description,Qty,ListPrice
FROM Customer C, Orders O, Items I, Parts P
WHERE
      (C.CustNo = O.CustNo)
AND   (O.OrderNo = I.OrderNo)
AND   (I.PartNo = P.PartNo)
ORDER BY Company, OrderNo, SaleDate, Description
```

Figyelem! Ha jelentésünkben valamilyen szempont szerint csoportosítunk, akkor a lekérdezésben az egy csoportba tartozó adatoknak egymás után kell következniük. Ezért rendezzük az adatokat cégnév, azon belül pedig rendelésazonosító szerint. A dátum szerinti rendezés már csak azt befolyásolja, hogy a megrendelések tételei milyen sorrendben jelenjenek meg.

Helyezzünk el az űrlapon egy *QuickRep* komponenst (*qrVevoRend*). *DataSet* jellemzőjét állítsuk a lekérdezésre (*qVevoRendTetelek*).

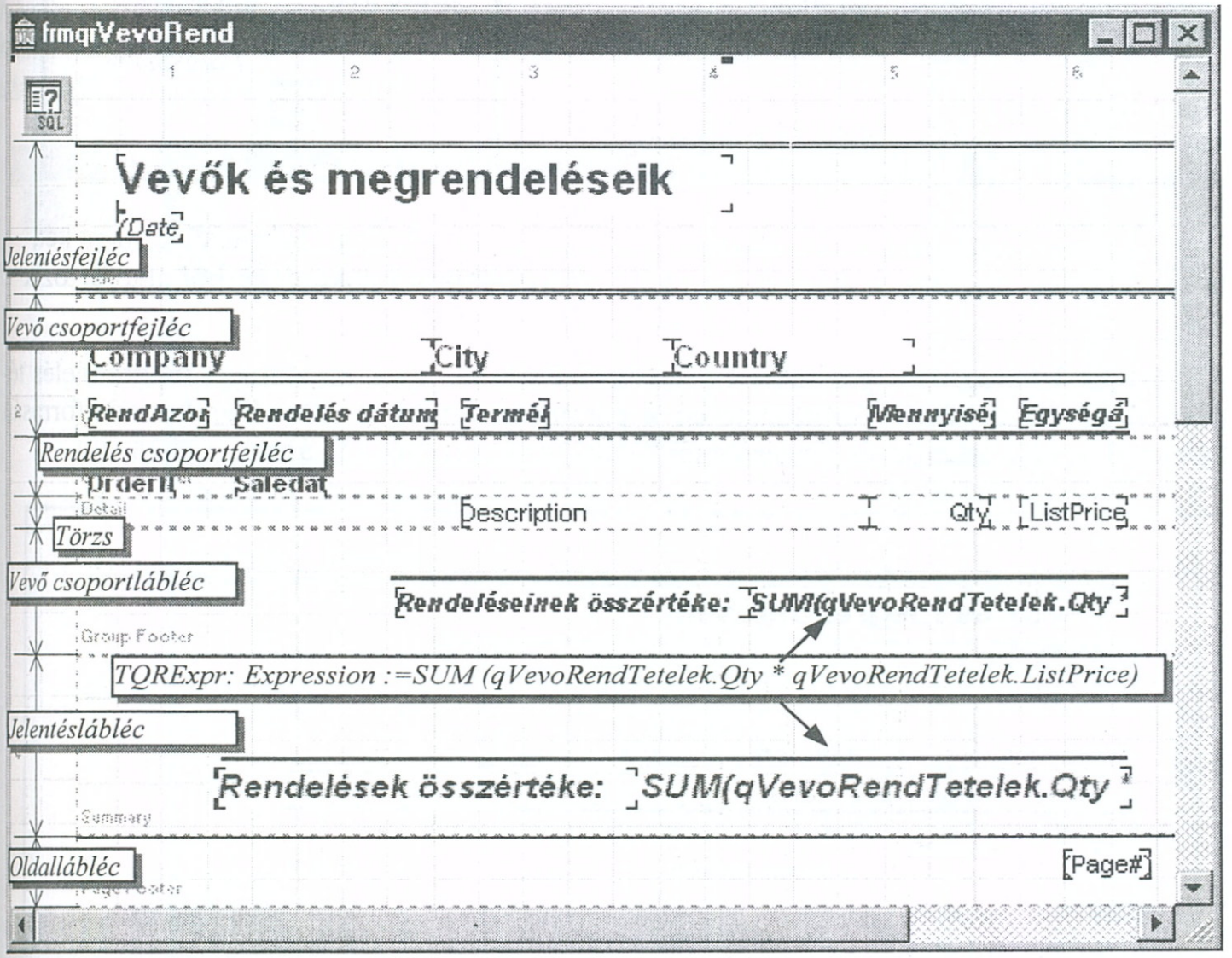
Jelentésünk most hét szakaszból áll (13.8. ábra): jelentésfejléc (*TQRBand*), vevő csoportfejléc (*TQRGroup*), rendelés csoportfejléc (*TQRGroup*), törzs (*TQRBand*), vevő csoportlábléc (*TQRBand*), jelentéslábléc (*TQRBand*) és az oldallábléc (*TQRBand*).

A vevő csoportfejléc komponensnél állítsuk be a csoportosítás szempontját a *'Company'* mezőre, a *FooterBand* jellemzőt pedig a vevő csoportlábléc komponensre.

A rendelés csoportfejlécben a csoportosítás az *'OrderNo'* mező értékei alapján történik.

Helyezzük el a szakaszokban a megjelenítendő információkat, amint ezt a 13.8. ábra is mutatja. Az *Egységárat* és *Rendelés összértékét* igazítsuk jobbra, annak érdekében, hogy a tizedesvesszők egymás alá kerüljenek. A számított értékeknél még formátumot is specifikálnunk kell: *Mask := 0.00 Ft.*

Figyeljük meg, mi történik, ha a vevő csoportláblécben megjelenő kifejezés *ResetAfterPrint* jellemzőjét nem állítjuk Igazra!



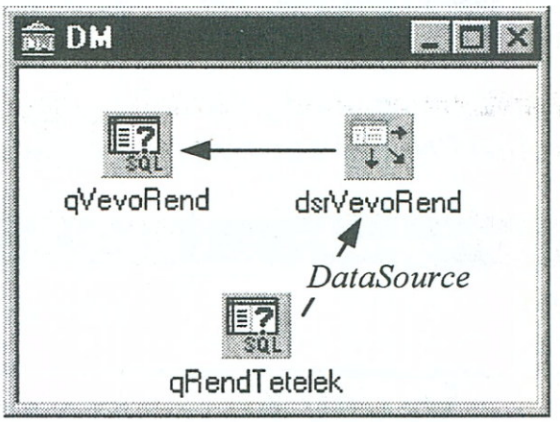
13.8. ábra. A kétszintű csoportváltásos lista megvalósítása két *TQRGroup* komponenssel

13.4.3.2 Második megoldás: beágyazott jelentés használata (TQRSubDetail komponenssel)

Ha minden vevőnél ki akarjuk számolni megrendeléseinek számát, valamint ezek összértékét is (13.10. ábra), akkor jelentésünkben két törzsszakaszra van szükségünk. Az egyikben a vevő megrendeléseit kell megjelenítenünk, hogy összeszámolhassuk ezeket a *Count* függvénnyel. A másik törzsszakaszban az adott megrendelés tételeit kell feltüntetnünk,

ezek alapján fogjuk a megrendelés összértékét kiszámolni. A két törzsszakasz két külön lekérdezésből fog „táplálkozni".

```
{qVevoRend}
SELECT  Company, Addr1, City, Country,
        OrderNo, Saledate
FROM Customer C, Orders O
WHERE (C.CustNo = O.CustNo)
ORDER BY Company, SaleDate

{qRendTetelek}
SELECT Description, Qty, ListPrice
FROM Items,Parts
WHERE
      (Items.OrderNo = :OrderNo)
AND   (Items.PartNo = Parts.PartNo)
```



13.9. ábra. Jelentésünk most két, kapcsolt lekérdezésből „táplálkozik"

A második lekérdezés – paraméterének köszönhetően – csak az aktuális megrendelés tételeit fogja tartalmazni. Természetesen a paraméter értéke a *qRendTetelek* adatforrásán keresztül az első lekérdezés *OrderNo* mezőjéből származik (13.9. ábra).

Vevők és megrendeléseik

1998.03.17.

Action Club **Sarasota** **US**

RendAzon	*Rendelés dátuma*	*Termék*	*Mennyiség*	*Egységár*
1014	1988.05.25.			
		Krypton Flashlight	3	44,95 Ft
1029	1988.07.18.			
		Regulator System	46	430,00 Ft
		Safety Knife	8	41,00 Ft
1038	1988.08.26.			
		Depth/Pressure Gauge Console	54	188,00 Ft
1129	1993.10.19.			
		Flashlight (Rechargeable)	4	169,95 Ft
		95.1 cu ft Tank	1	325,00 Ft

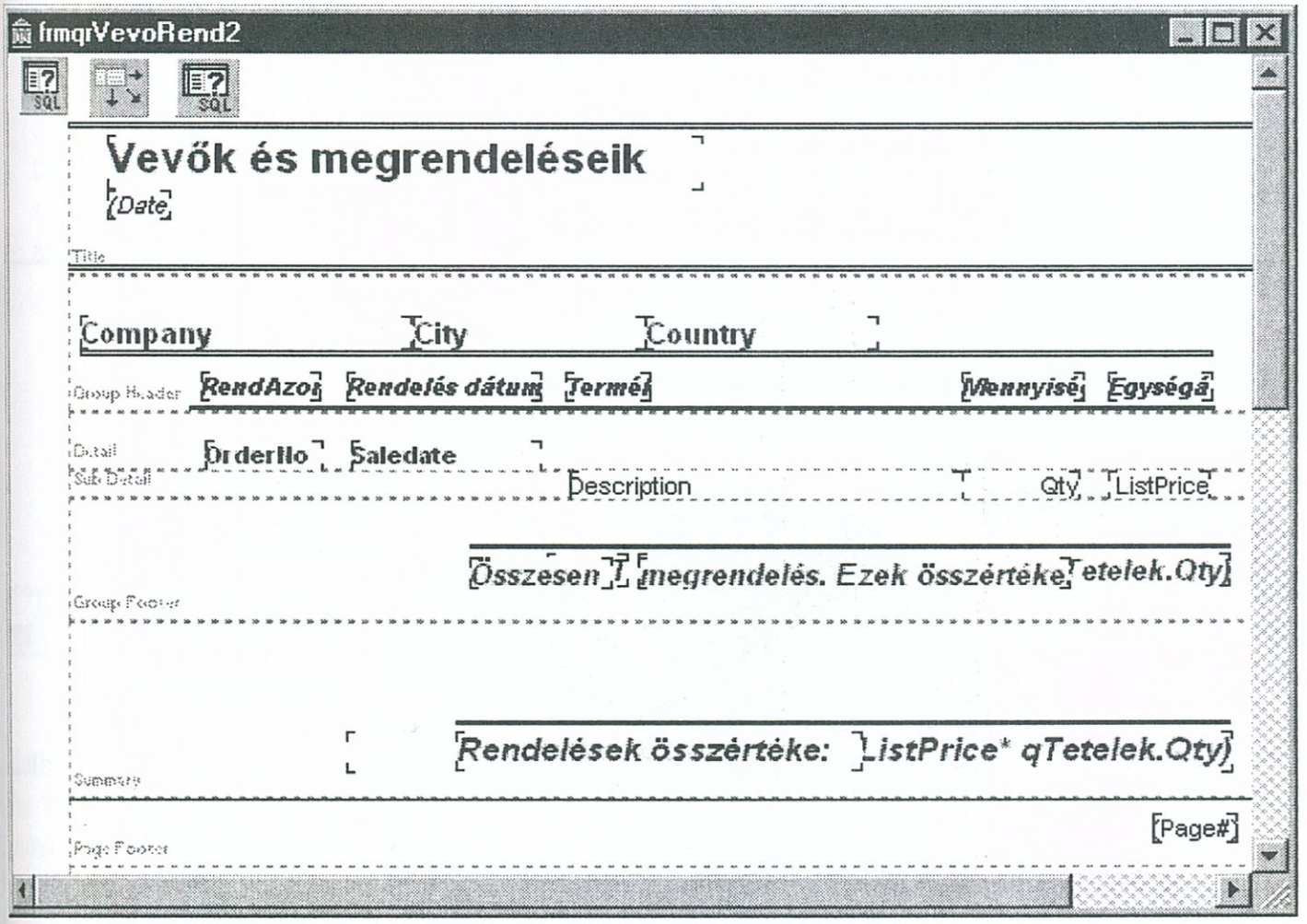
Összesen 4 megrendelés. Ezek összértéke: 31399,65 Ft

13.10. ábra. Kétszintű csoportváltásos lista egy *TQRSubDetail* komponenssel (Most már a megrendelések számát is meg tudjuk jeleníteni.)

Hozzunk létre egy új űrlapot (*frmqVevoRend2*). Helyezzük el rajta a két lekérdezést (tehetjük külön adatmodulra is, lásd 13.9. ábra). Az új jelentéskomponens neve legyen *qVevoRend2*.

Újból hét szakaszból rakjuk össze jelentésünket: jelentésfejléc (*TQRBand*), vevő csoportfejléc (*TQRGroup*), törzs (*TQRBand*), beágyazott törzs (*TQRSubDetail*), vevő csoportlábléc (*TQRBand*), jelentéslábléc (*TQRBand*) és az oldallábléc (*TQRBand*).

A beágyazott jelentés *DataSet* jellemzőjét állítsuk a második lekérdezésre (*qRendTetelek*), *Master* jellemzőjét pedig a *qVevorend2* jelentésre. Ezzel mondjuk meg azt, hogy ez egy alárendelt, beágyazott jelentéstörzs.



13.11. ábra. A kétszintű csoportváltásos lista megvalósítása *TQRSubDetail* komponenssel

A vevő csoport láblécében most két kifejezés eredményét is megjelenítjük. Az elsőben a vevő megrendeléseinek számát számoljuk ki. Itt tehát a *Count* függvénnyel a (főjelentés) törzs rekordjait kell összeszámolnunk, ezért a kifejezés *Master* jellemzőjét állítsuk a főjelentésre (*qrVevoRend2*). A második kifejezésben a vevő megrendeléseinek összértékét szeretnénk megjeleníteni. Ehhez a *SUM(qRendTetelek.Qty*qRendTetelek.ListPrice)* képlet szükséges, mely a beágyazott törzs rekordjaira vonatkozik. Emiatt állítsuk a kifejezés *Master* jellemzőjét a beágyazott jelentésre.

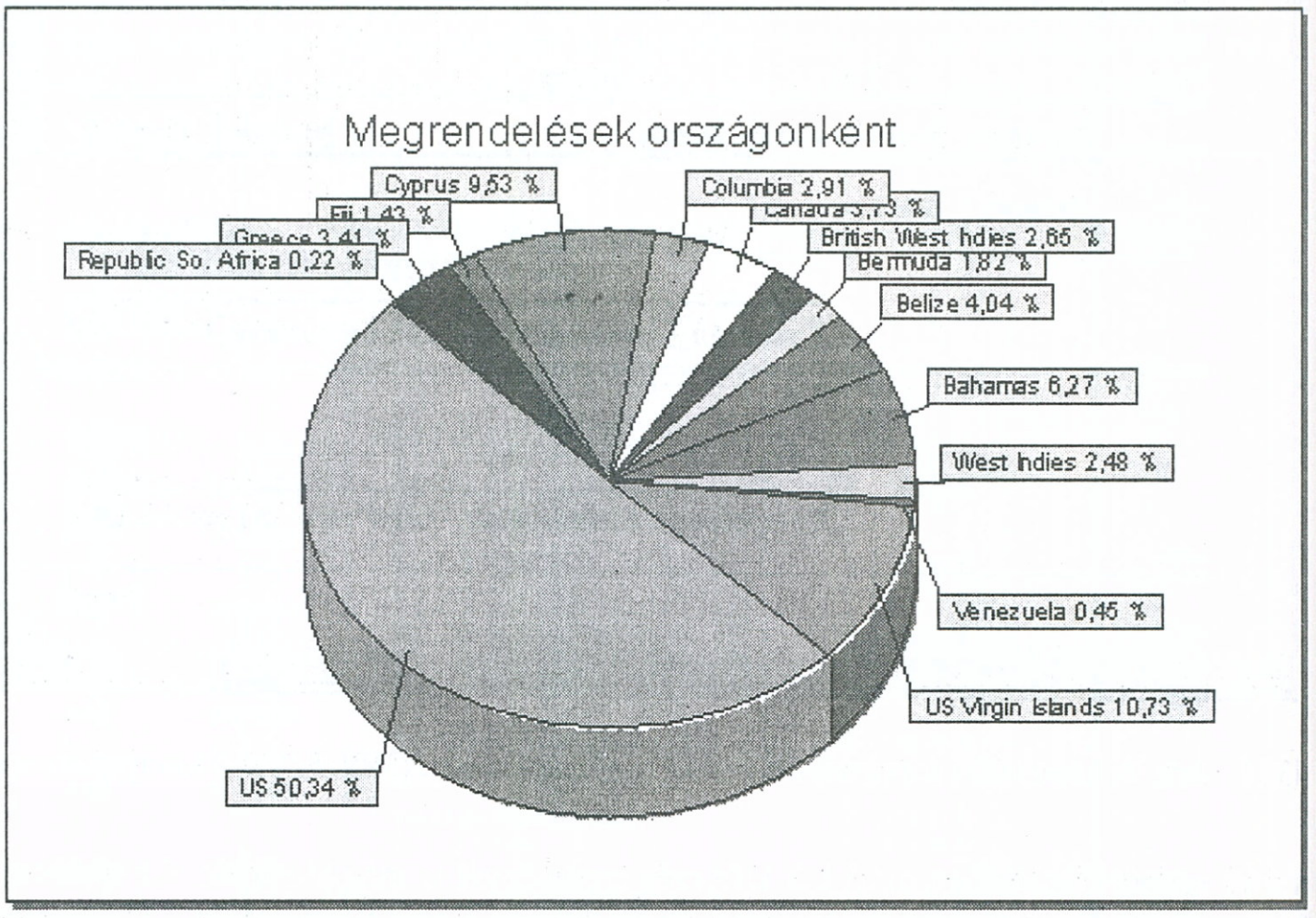
Próbáljuk ki a jelentést! Azt vesszük észre, hogy a második megoldással készített jelentés lassabban készül el, mint az első. Ez a paraméteres lekérdezés (*qRendTetelek*) miatt van, hiszen ez minden egyes megrendelés esetén újra lefut.

13.4.4 Diagramok

Készítsünk egy diagramot az országonkénti megrendelések összértékének összehasonlítására. Ehhez használhatjuk a Delphi 3-ba már beépített *TQRChart* komponenst.

Hozzunk létre egy új űrlapot (*frmDiagram*). Helyezzünk el rajta egy *TQuery* komponenst (*qDiagram*), melynek *SQL* jellemzőjébe a következőket írjuk:

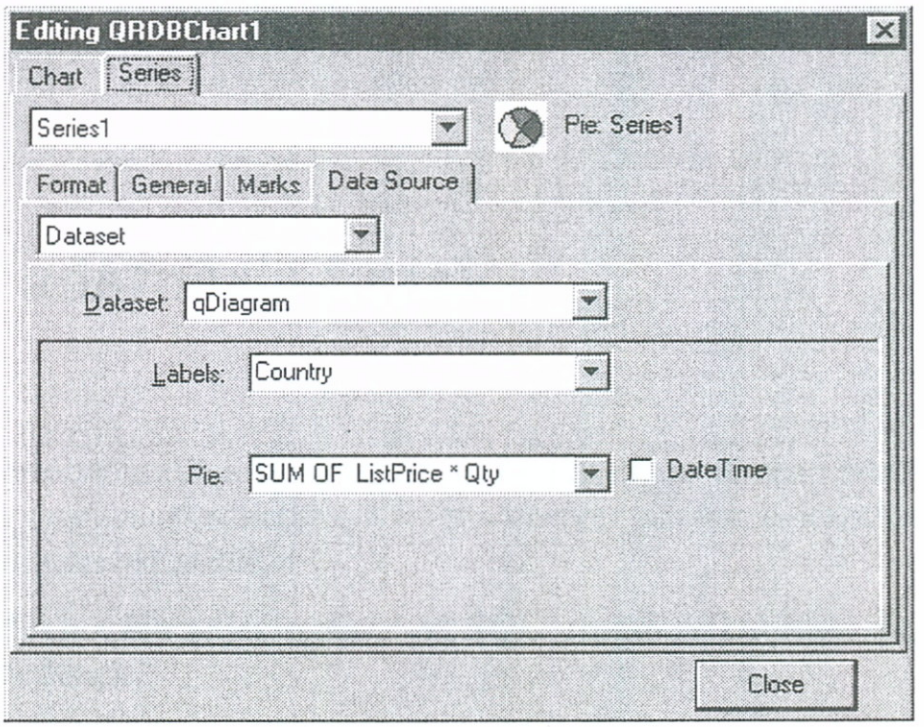
```
SELECT Country, SUM(ListPrice*Qty)
FROM Customer C, Orders O, Items I, Parts P
WHERE (C.CustNo = O.CustNo)
      AND (O.OrderNo = I.OrderNo)
      AND (I.PartNo = P.PartNo)
GROUP BY Country
```



13.12. ábra. A megrendelések összértéke országonként

A diagramot akár egy űrlapon, akár egy jelentésben is megjeleníthetjük. Most készítsünk egy jelentést. Helyezzünk el űrlapunkon egy *QuickRep* komponenst (*qrDiagram*). Egyetlen egy szakaszból fog állni, a jelentésfejlécéből. Helyezzünk el ebben egy *TQRChart* komponenst. Kattintsunk duplán a grafikonra, megjelenik a diagram-szerkesztő párbeszédablak. Végezzük el benne a következő beállításokat:

- A *Chart/Series* oldalon kattintsunk az *Add* gombra. Itt kell kiválasztanunk a diagram típusát. Példánkban kördiagramot szeretnénk készíteni (*Pie*).
- Váltsunk át a *Series/ DataSource* oldalra. Itt fogjuk a diagram adatforrását beállítani. Végezzük el a beállításokat, amint ezt a 13.13. ábra mutatja.



13.13. ábra. A grafikonszerkesztő párbeszédablak

Tanulmányozza a párbeszédablakban felkínált formázási lehetőségeket! Forgassa el a diagramot pár fokkal úgy, hogy az országneveket tartalmazó címkék olvashatók legyenek.

Adatbázisos diagramok készítésére használhatjuk még a Delphi 3 Client/Server változatába beépített *Decision Cube* komponenscsaládot, vagy más, erre a célra kifejlesztett *„third party"* komponenseket is.

Feladatok

Könyvnyilvántartónk kiegészítése jelentésekkel
(💾 13_KONYVTARFOLYTATAS\PKONYVTAR.DPR)
A 12. fejezetben lekérdezésekkel is feldúsított könyvnyilvántartó alkalmazáshoz most készítse el a szükséges jelentéseket:

- Kiadók, témák, írók listája: egyszerű jelentések (csak az adott táblában szereplő adatokkal)
- Könyvek listája: könyvenként tüntesse fel ennek adatait és szerzőit.

14. Kliens/szerver adatbázis-kezelés

Az előző fejezetekben láthattuk, kipróbálhattuk a fájl-szerver architektúrában történő adatbázis-kezelést. Ebben a fejezetben megismerkedhetünk a kliens/szerver adatbázis-kezelés sajátosságaival egy rövid mintafeladaton keresztül.

Ebben a fejezetben az adatbázis létrehozása teljes egészében SQL utasításokkal történik, emiatt SQL ismereteink elengedhetetlenek.

14.1 Feladatspecifikáció

Készítsünk egy mini hallgatói-nyilvántartást, melyben tároljuk a hallgatók adatait, a tantárgyakat, valamint a hallgatók jegyeit a különböző tantárgyakból. Adatszolgáltatóként most a Local Interbase Servert fogjuk használni, így a teljes adatbázist SQL utasításokkal fogjuk létrehozni a tábláktól kezdve, a triggerekig és tárolt eljárásokig. Erre a Windows ISQL segédprogramot fogjuk használni.

Az Interbase adatbázisra épített Delphi alkalmazásban lehessen a hallgatók adatait és jegyeit karbantartani, valamint lehessen megjeleníteni a tantárgyi átlagaikat. A programnak figyelnie kell azt is, hogy a hallgatók pillanatnyilag hány tantárgyból állnak bukásra. Ha egy hallgató bukásra áll, akkor a programnak valamilyen módon (hanggal vagy képpel) figyelmeztetnie kell erre a felhasználót (aki majd figyelmeztetheti például a Pistikét, vagy talán inkább a szüleit?).

Megoldás (🖫 Adatbázis: ADATOK\JEGYEK\JEGYEK.GDB
Alkalmazás: 14_JEGYEK\JEGYEK.DPR)

A megoldást most is az adatbázis megtervezésével és implementálásával kezdjük. Az adatbázis létrejötte után készítjük majd el Delphi alkalmazásunkat.

14.2 Az adatbázis megtervezése

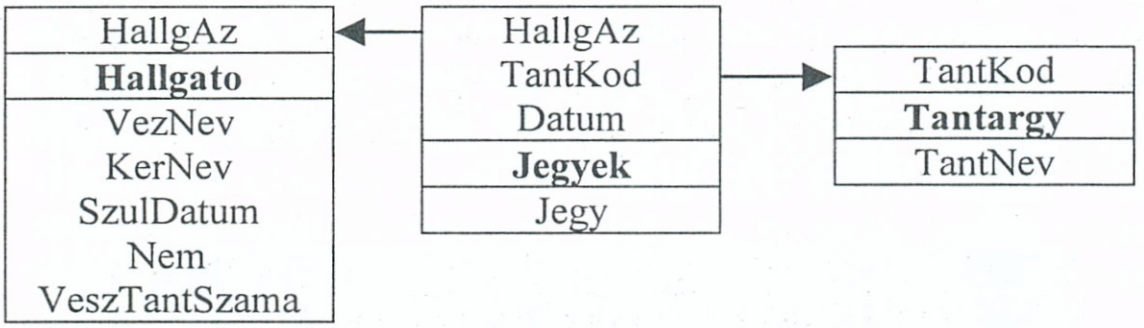
A követelményrendszert megvizsgálva tapasztaljuk, hogy a következő adatokat kell tárolnunk[1]:

- **Tantárgyakról**:
  - **TantKod**: a tantárgy kódja; 4 karakteres mező, például PRG1, MAT1, MAT2...
  - **TantNev**: a tantárgy leírása; 30 karakteres mező, például Programozás alapok...
- **Hallgatókról**:
  - **HallgAz**: a hallgató azonosítója; értékét a program automatikusan generálja (számláló típusú)
  - **VezNev**: a hallgató vezetékneve; 20 karakteres mező
  - **KerNev**: a hallgató keresztneve; 40 karakteres mező
  - **SzulDatum**: születési dátuma; dátum típusú
  - **Nem**: a hallgató neme (csak a példa kedvéért tároljuk); Interbaseben nincs logikai típus, így ezt most szimulálnunk kell: legyen a *Nem* egy 1 karakteres mező, melynek értéke vagy 'F', vagy 'N'.
  - **VeszTantSzama** (veszélyes tantárgyak száma): azoknak a tantárgyaknak a száma, melyekből a hallgató bukásra áll. Ez egy kiszámítható érték, melyet a feladatspecifikáció szerint figyelnünk kell. Ha nem tárolnánk az adatbázisban, akkor minden egyes hallgató megjelenítésénél ki kellene számolnunk. Természetesen a számolást meg kellene ismételnünk minden újabb jegy felvitelénél, törlésénél és módosításánál.

    Ezzel szemben, ha ezt az értéket a *Hallgato* táblában tároljuk, akkor elég a számolást minden jegy felvitelnél, módosításnál, törlésnél elvégezni, a hallgató adatainak megjelenítésekor ez fölöslegessé válna. A második megoldás egyik előnye tehát az elsővel szemben az, hogy így a számítást ritkábban kell elvégeznünk. Ugyanakkor, ha egy tárolt értékről van szó, akkor annak kiszámolását automatizálhatjuk triggerek segítségével (lásd a 14.2.6. pontban), így a kód automatikusan és kikerülhetetlenül le fog futni az adatbázis-szerveren. Ez gyorsaságot, hatékonyságot, biztonságot jelent.
- A hallgatók **jegyeit** a különböző tantárgyakból egy külön, kapcsolótáblában tároljuk. Ennek mezői:
  - **HallgAz**: melyik hallgató,
  - **TantKod**: melyik tantárgyból,
  - **Datum**: mikor,
  - **Jegy**: milyen jegyet kapott. A jegy valós típusú (adjuk meg a lehetőségét annak, hogy a 3,5 jegyet is felvihessenek), de értékei csak 1 és 5 közöttiek lehetnek.

---

[1] Könyvünkben már eleve a fizikai adatmodellt közöljük, ezt természetesen megelőzte a teljes modellezési folyamat.



14.1. ábra. Az adatmodell

Következik az adatmodell implementációja. Adatbázisunkba most az alkalmazás-logikát is be tudjuk építeni, hiszen ezt az Interbase formátum támogatja a különböző megszorítások, triggerek, tárolt eljárások formájában. A következőkben tehát az alábbi adatbáziselemeket fogjuk létrehozni:

- Mezőtípusok:
  - DNem: a hallgató nemének
  - DTantKod: a tantárgykódnak
  - DJegy: a hallgatói jegyeknek
- Táblák: az adatmodellen feltüntetett táblák, kapcsolatokkal és indexekkel együtt
- Triggerek:
  - TBI_Hallgato: ez fogja az újonnan felvitt hallgatók azonosítóját generálni
  - TAI_Jegyek, TAU_Jegyek, TAD_Jegyek: a VeszTantSzama mező aktualizálására
- Tárolt eljárások:
  - EgyHallgatoAtlagai: a paraméterként kapott azonosítóval rendelkező hallgató tantárgyi átlagainak kiszámolására (ezeket meg kell tudnunk jeleníteni a kliens oldali alkalmazásban, lásd feladatspecifikáció)

Mindezek előtt azonban létre kell hoznunk a fizikai adatbázist, melyben a következő lépésekben elhelyezzük majd az előbbiekben felsorolt elemeket.

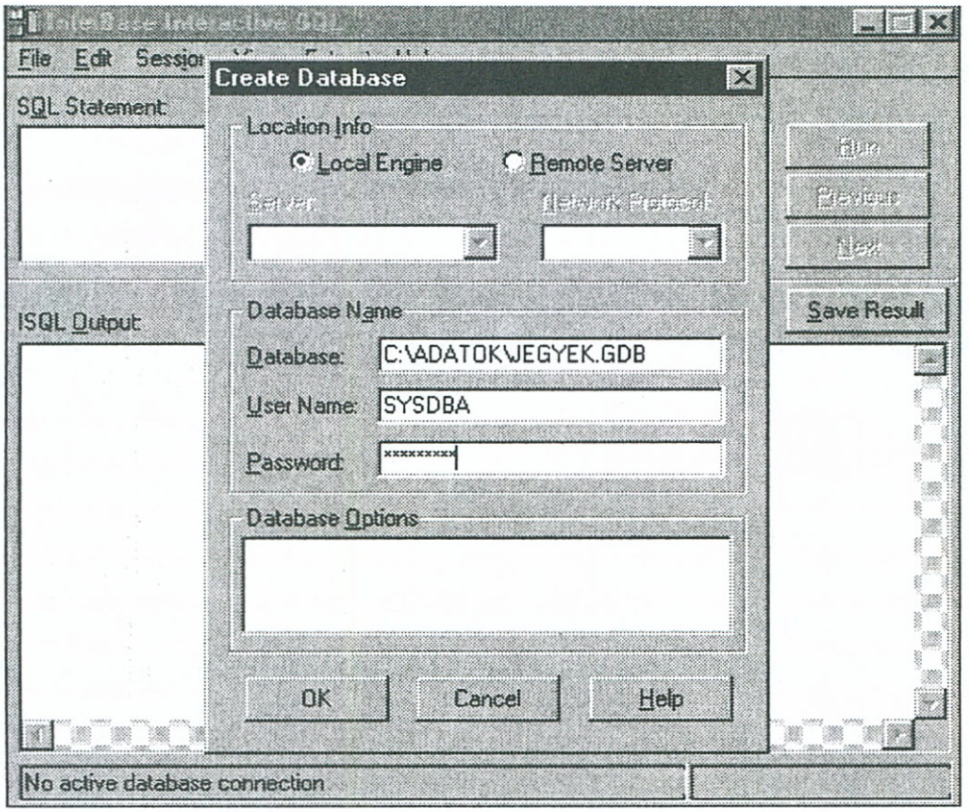
14.2.1 A fizikai adatbázis létrehozása

Lépések:

- Hozzunk létre egy könyvtárat az adatbázis számára (Például C:\ADATOK).
- Indítsuk el az Interbase Windows ISQL segédprogramot.
- Hívjuk meg a *File/New Database* menüpontot, majd adjuk meg az új adatbázis adatait (lásd 14.2. ábra).

  A felhasználó (*User Name*) = SYSDBA, a jelszó (*Password*) = masterkey.

Minden adatbázis-szerver ún. felhasználók (*users*) segítségével szabályozza az adatkezelési jogosultságokat. Minden adatbázis-szerver esetén létezik egy adminisztrációs jogokkal felruházott felhasználó, ez az Interbase esetén a *SYSDBA* nevet viseli (MSSQL-ben *SA*, Oracleben *DBA*). Jelszava alapértelmezés szerint *masterkey*, éles rendszerekben ezt természetesen le kell cserélni.



14.2. ábra. Adatbázis létrehozása az Interbase Windows ISQL segédprogrammal

Még ne zárjuk be a Windows ISQL programot, ebben fogjuk sorban a megfelelő SQL utasításokkal létrehozni a táblákat, triggereket... Az SQL utasításokat begépelhetjük, és lefuttathatjuk egyenként, vagy összegyűjthetjük mindet egy SQL szkriptállományba, és egyszerre végrehajthatjuk a *File/Run an SQL Script...* paranccsal.
Aki gépközelben van, annak azt javaslom, próbálja meg egyenként begépelni az utasításokat. Aki egyszerre szeretné az utasításokat lefuttatni, az használhatja az adatbázis mellett található JEGYEK.SQL szkriptállományt.

14.2.2 A mezőtípusok (*Domains*) létrehozása

A *Domain* egy felhasználó által létrehozott adatbázisszintű típus, akárcsak a *Type* szóval bevezetett típusok a Pascalban.
Például

```
CREATE DOMAIN DTantKod AS CHAR(4) NOT NULL;
CREATE DOMAIN DJegy AS NUMERIC(3, 2)
            CHECK(VALUE BETWEEN 1 AND 5);
```

```
CREATE DOMAIN DNem AS CHAR(1)
          CHECK(VALUE IN ('F', 'N'));
```

Akárcsak a Pascalban, Interbaseben is, ha egy típust több helyen (több meződefiníciónál) használni szeretnénk, akkor érdemes létrehozni számára egy külön típusazonosítót a *CREATE DOMAIN* utasítással. Egy típusba beépíthetünk kezdőértékeket (*DEFAULT*) és különböző értékellenőrzéseket (*constraints, CHECK*). Ha pedig később változtatni akarjuk, akkor elég a típust átírni, nem kell az összes mezőnél elvégezni a módosítást.
Példánkban a következő három típust hozzuk létre:

- **DTantKod**: a tantárgykódok számára (például PRG1); mindig pontosan 4 karakterhelyet foglal, még akkor is ha a felhasználó egy MAT kódú tantárgyat venne fel. Ezzel szemben a *VARCHAR(4)* típusú mező maximum 4 karaktert foglalna, de a tényleges méretét a pillanatnyi értéke szabná meg. A *CHAR* típus adatok helyigényesebbek a *VARCHAR* típusúaknál, de ugyanakkor a karakterlánc műveleteket gyorsabban lehet rajtuk végrehajtani. Tehát akkor ajánlott a *CHAR* típusok használata, amikor a mező értékei azonos hosszúságúak (például *Irányítószám*, de semmi esetre sem *SzemélyNév*).

  A *NOT NULL* azt jelenti, hogy az ilyen típusú mezők kitöltése kötelező.

- **DJegy**: a hallgatók jegyeinek (például 3.12); valós típusúnak definiáljuk, értékei csak 1 és 5 közöttiek lehetnek.

- **DNem**: a hallgató nemének típusa; a *DNem* egy karakteren tárolja a hallgató nemét: 'F' ⇨ Férfi, 'N' ⇨ Nő.

  A típusban definiált megszorítások (*CHECK*) a mezőérték tárolásakor értékelődnek ki. A nem megfelelő értékek nem kerülnek be az adatbázisba, és erre egy hibaüzenet is figyelmeztet. Annak érdekében, hogy a nemnél a kis 'f' és 'n' betűket is fogadja el a rendszer, a bevitt karaktert automatikusan (egy trigger segítségével) nagybetűssé fogjuk alakítani. Így, mire tárolásra kerül a sor, a *Nem* mezőben mindig nagybetűs érték lesz. (Megvalósítását egy kicsit később részletezzük.)

  Bizonyára sok kedves Olvasó fejében megfordul, hogy mindezt a kliens oldali Delphi alkalmazásban milyen egyszerűen meg lehetne valósítani. Ez viszont már az alkalmazás-logika implementálási helyének a kérdésköre, lásd a 14.3.2. pontban.

14.2.3 A táblák létrehozása

Hozzuk létre sorban a táblákat:

```
CREATE TABLE Hallgato (
    HallgAz INTEGER NOT NULL,        {NOT NULL ⇨ kötelező kitölteni}
    VezNev VARCHAR(20) NOT NULL,
    KerNev VARCHAR(40) NOT NULL,
    SzulDatum DATE,
    Nem DNem,
    VeszTantSzama INTEGER,
  PRIMARY KEY (HallgAz));            {PRIMARY KEY ⇨ elsődleges kulcs}
```

```
{A HallgAz mezőnek számláló típusúnak kellene lennie, de mivel
Interbaseben nincs ilyen típus, Integer-nek deklaráljuk. Az értékeit a
program fogja generálni az ún. generátorok segítségével, lásd később.}

CREATE TABLE Tantargy (
      TantKod DTantKod,
      TantNev VARCHAR(30) NOT NULL,
   PRIMARY KEY (TantKod));

CREATE TABLE Jegyek (
      HallgAz INTEGER NOT NULL,
      TantKod DTantKod,
      Datum DATE DEFAULT 'NOW' NOT NULL,
      {DEFAULT ⇨ kezdőérték, 'NOW' ⇨ a mai dátum }
      Jegy DJegy NOT NULL,
   PRIMARY KEY (HallgAz, TantKod, Datum));

{A hivatkozási integritási kapcsolatok megadása}
ALTER TABLE Jegyek ADD  FOREIGN KEY (HallgAz)
                        REFERENCES Hallgato(HallgAz);

ALTER TABLE Jegyek ADD  FOREIGN KEY (TantKod)
                        REFERENCES TANTARGY(TantKod);

{Másodlagos indexek létrehozása:
CREATE INDEX <Indexnév> ON <Táblanév> (<IndexeltMezők>) }
CREATE INDEX HallgNevIDX ON Hallgato (VezNev, KerNev);
```

Gépelje be, majd futtassa le egyenként a táblalétrehozó utasításokat! Ha ezek után meghívja az *Extract/SQL Metadata for Table* menüpontot, akkor meggyőződhet a táblák létezéséről.

14.2.4 A generátorok létrehozása

Mint már említettem, Interbaseben nem létezik számláló típus, azonban van lehetőség ennek szimulálására. A lényege az, hogy első lépésben bevezetünk egy generátort (*GENERATOR*). Ez olyan, mint egy változó, melynek kezdőértéket is adhatunk. A mi példánkban a generátor a következő lesz:

```
CREATE GENERATOR GEN_HallgAz;
{Ha nullától különbözőre szeretnénk beállítani, akkor a ezt következő-
képpen tegyük: SET GENERATOR GEN_HallgAz 100;}
```

Ezek után írnunk kell egy triggert, mely minden újabb hallgató felvitelénél eggyel növeli a generátor értékét, majd az új hallgató azonosítóját erre az értékre állítja be (bővebben lásd később).

14.2.5 Pár szó a triggerekről és tárolt eljárásokról

A **triggerek** (*Triggers*) és **tárolt eljárások** (*Stored Procedures*) **kiterjesztett SQL-ben írt rutinok**. A trigger egy eseményvezérelt eljárásnak felel meg (így hívása automatikus), míg a tárolt eljárást tételesen meg kell hívnunk, akár a kliens oldali programból is (a *TStoredProc* komponens segítségével). A triggert tehát egyértelműen egy adott tábla egy bizonyos eseményéhez rendeljük. Az események a következők lehetnek: *Before Insert, After Insert, Before Update, After Update, Before Delete, After Delete*.

A triggerek és tárolt eljárások előnyei:

- Utasításaik a szerveren futnak le, a klienshez csak az esetleges eredmények „utaznak le", ezzel is csökkentve a hálózati forgalmat, és ugyanakkor növelve az adatok biztonságát.
- Ha egy triggerbe adatellenőrzést építünk be, akkor ez sehonnan sem kerülhető ki: akár Delphi programból, akár itt, a Windows ISQL-ből szeretnénk egy értéket felvinni, lefut a trigger – és benne az adat ellenőrzése –, ezzel meggátolva az adatok elrontását.
- A triggereket és tárolt eljárásokat már az adatbázis megtervezésekor megírjuk, így később minden adatbázis felhasználója képes lesz ezeket meghívni anélkül, hogy a rutinokat neki kellene megírnia, és elküldenie a hálózaton.
- A tárolt eljárás végrehajtása gyorsabb, mint az ezt alkotó SQL utasítások lefuttatása, mivel a tárolt eljárás leellenőrzött, optimalizált és előfordított állapotban kerül tárolásra az adatbázisban.
- A tárolt eljárások használatának egy másik előnye az, hogy segítségükkel a jogosultságokat is kényelmesebben és hatékonyabban lehet szabályozni. Így csak a tárolt eljárás végrehajtásának jogát kell szabályoznunk, nem pedig az általa használt táblák, nézetek bonyolult jogosultságrendszerét. Persze ez csak akkor hatásos, ha az érintett táblák, nézetek közvetlen elérését megtiltjuk, vagyis az adatokhoz csak a tárolt eljáráson keresztül férhetnek hozzá a felhasználók, közvetlenül nem.

A triggereket adatellenőrzésekre, kezdőérték beállításokra, bizonyos származtatott és tárolt mezők kiszámolására szoktuk használni. Tárolt eljárásként a nagy adatmennyiséget feldolgozó és a jogosultságokhoz kötött tevékenységeket szoktuk megírni, mint például egy hallgató-nyilvántartásban a félév végi zárás, jegyek átlagolása... **Általános irányelvként igyekezzünk minél több műveletet tárolt eljárás formájában megvalósítani** a felsorolt előnyök miatt.

Mindezekre látunk konkrét példákat is a következő pontokban.

14.2.6 A triggerek létrehozása

Példánkban először is szükség van egy olyan triggerre, mely a hallgatók azonosítóját automatikusan generálja. Az új hallgatóazonosító generálása egy új hallgató felvitelénél esedékes, így a triggert a *Hallgato* tábla *Before Insert* eseményére építjük:

```
CREATE TRIGGER TBI_Hallgato FOR Hallgato
BEFORE INSERT
AS
BEGIN
   new.VeszTantSzama = 0;
   new.Nem=UPPER(new.Nem);
   new.HallgAz=GEN_ID(Gen_HallgAz, 1);
END
```

Az eseménykezelőkben minden mezőnek két értékére hivatkozhatunk: *old.mezőnév* a régi, még változtatás előtti értékét jelenti, a *new.mezőnév* pedig a frisset.

Triggerünk lefut minden újabb hallgató felvitele előtt (még a tényleges tárolása előtt). Ekkor a *VeszTantSzama* mezőt lenullázzuk, mivel ez a hallgató még biztosan nem bukik semmiből sem. Ezek után a hallgató neménél beütött karaktert nagybetűssé alakítjuk; így a kis 'f' és 'n' betűkből nagy 'F' és 'N' lesz. Tároláskor az adatbázis csak az 'F' és 'N' betűket fogadja be, de ennek a cselnek köszönhetően a felhasználó a kis 'f' és 'n' betűket is használhatja.
Végül értéket adunk a *HallgAz* mezőnek is. A *GEN_ID* függvény megnöveli eggyel az első paramétereként kapott generátor értékét, és ezt adja vissza függvényértékként. A generátor növelésével biztosítjuk azt, hogy a következő hallgató eggyel nagyobb azonosítót kapjon.

A trigger lefutását a felhasználó nem veszi észre, nincs róla tudomása, és nincs beleszólási joga sem. Ez a nagyszerű a triggerekben!

Más triggerekre is szükségünk van. Minden hallgatónak figyelnünk kell a jegyeit: egy újabb jegy felvitelénél, meglévő jegy módosításánál és törlésénél aktualizálnunk kell a *VeszTantSzama* mező értékét. Ennek érdekében három triggert kell írnunk: a Jegyek tábla *After Insert*, *After Update* és *After Delete* eseményeire. Azért használjuk az *After*... eseményeket (és nem a *Before*...-t), mert a számolt értéknek a felvitel, módosítás és törlés megtörténése utáni állapotot kell tükröznie.

```
{Kiszámoljuk sorban a hallgató tantárgyankénti összes jegyeinek átlagát; ha egy tantárgyból az átlaga 1.5 vagy annál kevesebb, akkor növeljük a TantSzam gyűjtőben a veszélyeztetett tantárgyakat. Legvégül beírjuk a Hallgató tábla VeszTantSzam mezőjébe a kiszámolt értéket. Minden számolás az aktuális hallgatóra vonatkozik, azaz akinek az azonosítója = new.HallgAz.}

CREATE TRIGGER TAI_Jegyek FOR Jegyek
ACTIVE AFTER INSERT POSITION 0
AS
   DECLARE VARIABLE Atlag DECIMAL(3,2);
```

```
  DECLARE VARIABLE TantSzam INTEGER;
BEGIN
  Atlag = 0;
  TantSzam = 0;
  FOR   SELECT AVG(Jegy)
        FROM Jegyek
        WHERE (HallgAz = New.HallgAz)
        GROUP BY TantKod
        INTO :Atlag
  DO
     IF (Atlag <= 1.5) THEN
        TantSzam = TantSzam + 1;

  UPDATE Hallgato
  SET VeszTantSzama = :TantSzam
  WHERE HallgAz = New.HallgAz;
END
```

A *„FOR SELECT... DO...”* utasítás végrehajtja a *SELECT* lekérdezést, és minden egyes eredményrekordra végrehajtja a *DO* utáni utasításokat. Konkrét esetünkben kiszámolja a hallgató tantárgyi átlagát, ennek értékét elhelyezi az *Atlag* lokális változóba (*INTO :Atlag*), majd végrehajtja a *DO* utáni utasítást: ha az átlag 1,5-nél kisebb, akkor inkrementáljuk a *TantSzam* gyűjtőt. (A használható SQL utasításokat lásd a Windows ISQL segédprogram súgójában: SQL Statement Reference.)

Triggerünket SQL-esebben így is megírhattuk volna:

```
CREATE TRIGGER TAI_Jegyek FOR Jegyek
ACTIVE AFTER INSERT POSITION 0
AS
BEGIN
  UPDATE Hallgato SET VeszTantSzama =
    (SELECT COUNT (*) FROM Tantargy
     WHERE TantKod IN
       (SELECT TantKod
        FROM JEGYEK
        WHERE HallgAz = new.HallgAz
        GROUP BY TantKod
        HAVING AVG(Jegy) <= 1.5)
    )
  WHERE HallgAz = new.HallgAz;
END
```

Figyeljük meg triggerünk definíciójában a *POSITION 0*-t. Az Interbase megengedi, hogy egy tábla egy adott eseményére több triggert is beépítsünk. Ezek lefutásának sorrendje a *POSITION*-ban megadott számtól függ.

Ugyanezt kell végrehajtanunk a jegyek tábla módosítása, valamint a táblából való törlés után is. (Folytassuk a „procedurális elménkhez" közelebb álló első változattal. A második SQL-esebb megoldás halmazorientált, és talán az SQL terén nem annyira jártas Olvasók számára kicsit nehezebben tekinthető át.):

```
CREATE TRIGGER TAU_Jegyek FOR Jegyek
ACTIVE AFTER UPDATE POSITION 0
AS
   DECLARE VARIABLE Atlag DECIMAL(3,2);
   DECLARE VARIABLE TantSzam INTEGER;
BEGIN
   Atlag = 0;
   TantSzam = 0;
   FOR   SELECT AVG(Jegy)
         FROM Jegyek
         WHERE (HallgAz = new.HallgAz)
         GROUP BY TantKod
         INTO :Atlag
   DO
      IF (Atlag <= 1.5) THEN
         TantSzam = TantSzam + 1;

   UPDATE Hallgato SET VeszTantSzama = :TantSzam
      WHERE HallgAz = new.HallgAz;
END

CREATE TRIGGER TAD_Jegyek FOR Jegyek
ACTIVE AFTER DELETE POSITION 0
AS
   DECLARE VARIABLE Atlag DECIMAL(3,2);
   DECLARE VARIABLE TantSzam INTEGER;
BEGIN
   Atlag = 0;
   TantSzam = 0;
   FOR   SELECT AVG(Jegy)
         FROM Jegyek
         WHERE (HallgAz = old.HallgAz)
         GROUP BY TantKod
         INTO :Atlag
   DO
      IF (Atlag <= 1.5) THEN
         TantSzam = TantSzam + 1;

   UPDATE Hallgato SET VeszTantSzama = :TantSzam
      WHERE HallgAz = old.HallgAz;
END
```

Törlés után annak a hallgatónak az adatait kell frissítenünk, akinek a jegyét éppen letöröltük. De ha most töröltük le, akkor honnan tudjuk az azonosítóját? A válasz az *old.HallgAz* mezőben rejlik. A *new.HallgAz* nem is létezik, hiszen kitöröltük, az *old.HallgAz* viszont még elárulja a törölt rekordbeli *HallgAz* mező értékét.

14.2.7 A tárolt eljárások létrehozása

Írjunk kezdetnek – csak a példa kedvéért – egy olyan tárolt eljárást, mely a paraméterként kapott hallgatónak, a paraméterként kapott tantárgyból kiszámolja az átlagát:

```
CREATE PROCEDURE Atlag (HallgAz INTEGER, TantKod VARCHAR(7))
RETURNS (Atlag NUMERIC(3,2))
AS
BEGIN
  SELECT AVG(Jegy)
  FROM Jegyek
  WHERE (HallgAz = :HallgAz) AND (TantKod = :TantKod)
  INTO :Atlag;
END;
```

Ha kíváncsiak vagyunk az 1-es azonosítójú hallgató átlagára a DLPI (Delphi) kódú tantárgyból, akkor a Windows ISQL-ben lefuttatjuk a tárolt eljárást a következő módon:

```
EXECUTE PROCEDURE Atlag(1,'DLPI')
```

Egy Delphi alkalmazásban ugyanezt a hatást érnénk el egy *spAtlag:TStoredProc* komponenssel.

```
With spAtlag Do
  begin
    {Jellemzőinek beállítása
    (legtöbbször ezt már tervezési időben megtesszük)}
    {Az álnév vagy az adatbáziskomponens}
    DatabaseName := 'DB';
    {A tárolt eljárás neve}
    StoredProcName := 'Atlag';

    {Paraméterek elkészítése, előfordítás}
    Prepare;
    {A bemeneti paraméterértékek beállítása}
    ParamByName('HallgAz').AsInteger:= 1;
    ParamByName('TantKod').AsString:= 'DLPI';
    {A tárolt eljárás végrehajtása}
    ExecProc;

    {Az eredmény a kimeneti paraméteréből olvasható ki}
    ShowMessage ('Az átlag ='+ ParamByName('Atlag').AsString);
  end
```

Most írjunk egy tárolt eljárást, mely a paraméterként kapott hallgatónak (*HallgAz*) kiszámolja, és visszaadja az átlagait (*TantNev, Atlag*). Ezt a tárolt eljárást már tényleg meg is fogjuk hívni a leendő Delphi alkalmazásunkból annak érdekében, hogy az aktuális hallgató átlagait megjeleníthessük. Ez az eljárás nem egy értékkel tér majd vissza, hanem egy rekordhalmazzal.

```
CREATE PROCEDURE EgyHallgatoAtlagai (HallgAz INTEGER)
RETURNS (TantNev VARCHAR(30), Atlag NUMERIC(3, 2))
AS
BEGIN
   FOR   SELECT TantNev, AVG(Jegy)
         FROM Tantargy, Jegyek
         WHERE   (Tantargy.TantKod = Jegyek.TantKod)
                 AND (Jegyek.HallgAz = :HallgAz)
         GROUP BY TantNev
         INTO :TantNev, :Atlag
   DO
      SUSPEND;
END
```

A *SELECT* lekérdezésben csoportosítunk a tantárgy szerint, minden egyes csoportban leválogatjuk a tantárgy nevét és átlagát, majd végrehajtjuk a *SUSPEND* utasítást. Ennek hatására az éppen leválogatott tantárgy neve és az átlag értéke bekerül az eredményhalmazba.

Az Interbase adatbázisok esetén a rekordhalmazzal visszatérő tárolt eljárások hívása nem *EXECUTE PROCEDURE* utasítással, hanem – a táblákból való lekérdezéshez hasonlóan – egy *SELECT*-tel történik (más adatbázis-szervereknél ez nem így van). Például, ha kíváncsiak vagyunk az 1-es azonosítójú hallgató átlagaira, akkor ezt a következőképpen kérdezhetjük le:

```
SELECT * FROM EgyHallgatoAtlagai(1);
```

Ha a tárolt eljárást egyszerűen *EXECUTE PROCEDURE* utasítással lefuttatnánk, akkor csak az első rekord értékeivel térne vissza.

```
EXECUTE PROCEDURE EgyHallgatoAtlagai(1) {💣}
```

Az ilyen tárolt eljárások lefuttatására, eredményének megtekintésére Delphiben a már ismert *TQuery* komponenst használjuk. Ennek SQL jellemzőjébe a következőket írnánk:

```
SELECT * FROM EgyHallgatoAtlagai (:HallgAz)
```

Természetesen a lekérdezés lefuttatása előtt be kell állítanunk a paraméter értékét. (Bővebben lásd később.)

14.2.8 A nézetek létrehozása

Ha már megismerkedtünk az Interbase adatbázis többi elemével, akkor a nézeteket (*views*) se hagyjuk ki. Ismerkedjünk meg röviden ezek fogalmával és kezelésével, annak ellenére, hogy konkrét alkalmazásunkban most nincs nézetekre szükség.

A nézet (*view*) egy vagy több tábla adatait gyűjti össze egy SELECT utasítással. Általában a statikus adatokat szoktuk nézetekben összegyűjteni, például a hallgatói névsort, tantárgyak listáját...

```
CREATE VIEW HallgNevsor (TeljesNev) AS
SELECT VezNev || ' ' ||  KerNev
FROM Hallgato
```

Amikor egy kliens alkalmazásban meg akarjuk jeleníteni a hallgatói névsort, megnyitjuk a lekérdezést (pontosabban a nézetre irányított táblakomponenst). Ilyenkor lefut a nézet SELECT lekérdezése, mely összegyűjti a kívánt adatokat. A lényeg tehát az, hogy nem kell kliens oldalon összeállítani és elutaztatni az adatbázis-szerverhez a lekérdezést, a nézet már eleve ott van, így most annak tartalmát kérdezzük le. Kliens oldalon a nézet úgy viselkedik, mint egy tábla, mintha ténylegesen adatot tartalmazna, holott csak az SQL lekérdezése tárolódik előfordított állapotban (mint a tárolt eljárások is). Delphiből a nézetek adatait *TTable* vagy *TQuery* komponensekkel érjük el.
A gyakran lekérdezendő adatokat érdemes már az adatbázis megtervezésekor egy nézetbe összegyűjteni, így ezeket kényelmesebben hozzáférhetővé tesszük. Természetesen egy nézet is csak akkor szerkeszthető, ha a lekérdezése is szerkeszthető lenne: egy táblán alapul, ha a nézetből kihagyott mezők értékeit nem kötelező kitölteni (ez egy nézetbe való rekord felvitelénél fontos), ha SELECT utasítása nem tartalmaz beágyazott lekérdezést, csoportosításokat, számított értékeket... Erre is az SQL-92 szabvány szabályai vonatkoznak.

14.2.9 A jogosultságok beállítása

Ha több felhasználója lenne programunknak, akkor előbb azokat fel kellene vennünk (ezt a Server Manager segédprogrammal tehetnénk meg), majd külön még a jogosultságaikat is be kellene állítanunk. A jogosultságok beállítása a Windows ISQL-ben történne a GRANT és REVOKE utasításokkal. Például:

```
GRANT SELECT ON HallgNevsor To Pistike WITH GRANT OPTION
{Ezzel a Pistike felhasználónak megengedjük, hogy a HallgNevsor nézetet lekérdezhesse (SELECT), valamint azt is, hogy saját jogosultságait másoknak is átadhassa (WITH GRANT OPTION)}
```

Konkrét feladatunkban egy felhasználó van csak, ez maga az adatbázis adminisztrátora, aki teljes jogokkal rendelkezik.

Tanulmányozza át még egyszer a teljes SQL szkriptállományt. Ennek érdekében töltse be a Delphibe. A Delphi 3.0 kódszerkesztője már felismeri és kiemeli az SQL utasításokat, ezzel növelve az áttekinthetőséget.

14.3 Az alkalmazás elkészítése

14.3.1 Az álnév létrehozása

Még a Delphi rendszer elindítása előtt létre kell hoznunk egy álnevet, mely az új adatokra fog mutatni. Indítsuk el a Database Explorert, hívjuk meg az *Object/New...* menüpontot. Állítsuk be az új álnév paramétereit a következők szerint:

```
{Neve = DBJegyek}
TYPE = InterBase
SERVER NAME = C:\ADATOK\JEGYEK.GDB {az adatbázis útvonala}
USER NAME = SYSDBA
ENABLE BCD = TRUE
```

Az **ENABLE BCD** beállítást a valós (NUMERIC és DECIMAL típusú) számok helyes használatának érdekében kell igazra állítanunk. Ezzel a BDE-be épített számok kezelésére irányuló optimalizálást kapcsoljuk ki, mely amúgy rosszul működne. A lényege az lenne, hogy a gyorsaság kedvéért az egész értékű valós számokat egész számként kezelje. Azonban a valóságban a törtszámokat is egészként kezeli, ami ahhoz vezet, hogy az adatmegjelenítési komponensek nem fogadnak el tizedes értékeket.

Kattintsunk duplán az álnévre. A rendszer bekéri a jelszót, majd megnyitja az adatbázist. Tallózhatunk a mezőtípusok, táblák, nézetek, eljárások... között. Sőt, ha kiválasztunk egy táblát, és az ablak jobb részében a *Data* fülre kattintunk, akkor a táblát fel is tölthetjük értékekkel.

Töltsük fel a *Tantargy* táblát néhány adattal. Delphi alkalmazásunkban másra fektetjük majd a hangsúlyt, ott nem tervezünk űrlapot a tantárgyak számára.

14.3.2 Pár szó az alkalmazás-logikáról (*Business Logic*)

Alkalmazás-logikának nevezzük az adatbázisra vonatkozó szabályok összességét, más szóval az adatbázis működési szabályait. Ezek között megtalálhatók a következők:

- Mezőszintű ellenőrzések: például a jegy csak 1 és 5 közötti lehet.
- Rekordszintű ellenőrzések: például egy tanfolyam kezdeti dátuma <= végdátuma.
- Hivatkozási integritás: például egy hallgató jegyét csak akkor lehet felvinni, ha a hallgató már szerepel a nyilvántartásban.
- Komplex ellenőrzések: például egy olvasó a könyvtárból maximum 4 könyvet kölcsönözhet ki.
- Egyéb adatbázis működését befolyásoló eljárások: például egy újabb jegy felvitelénél újraszámoljuk a *VeszTantSzama* mezőt.

Az alkalmazás-logika implementálásának módozatai

Elemezzük kicsit a kliens/szerver architektúra kínálta lehetőségeket: van egy adatbázisunk (adatbázis-szerveren), és van a kliens oldali alkalmazásunk. Az alkalmazás-logikát több szinten is beépíthetjük: implementálhatjuk az adatbázisban megszorítások (*constraints*), kapcsolatok, triggerek, tárolt eljárások formájában. Ha mindezek az adatbázisban vannak, akkor nem kerülhetők ki, függetlenül attól, hogy a Delphi alkalmazásból, vagy az Interbase Windows ISQL programból kezeljük az adatokat. Ezért jó, ha az alkalmazás-logikát az adatbázisba építjük.

A gyakorlat azonban azt mutatja, hogy vannak olyan szabályok, melyeket nem lehet, nem célszerű, vagy nagyon bonyolult lenne az adatbázisba építeni. Erre egy nagyon egyszerű példa a bemeneti maszk: egy telefonszámnak (999)-999-9999 formájúnak kell lennie. Ez is egy fontos információ, hiszen jó lenne, ha a telefonszám begépelésénél a program csak számjegyeket fogadna el, és összesen csak 10-et, a maszk szerinti bontásban. Ezt a szabályt hova építsük be? Ha a kliens oldali alkalmazásban helyezzük el (a mezőobjektum *EditMask* jellemzőjében), akkor a beállítást minden egyes telefonszám mezőn el kell végeznünk. Ha pedig változna a telefonszám formátuma, akkor mindenhol át kellene írnunk. Természetesen egy másik alkalmazásban ugyanúgy el kellene végeznünk egyenként a beállításokat. Ez egy kényelmetlen módszer lenne.

A problémára a megoldást az **adatszótárak**[2] (***Data Dictionary***) jelentik. A Database Explorer programban adatszótárt hozhatunk létre egy adatbázis számára, melyben mezőszinten beállíthatók a különböző megjelenítési és működési opciók: a mező formátuma, bemeneti maszkja, címkéje stb. sőt még az is, hogy a mezőnek a leendő Delphi űrlapon milyen komponens feleljen meg. Például a *Nem* mező ne szerkesztődobozban jelenjen meg, hanem egy választógomb-csoport formájában; úgyszintén a hallgató azonosítója ne legyen szerkeszthető, tehát ennek a *TDBText* komponens feleljen meg (az alapértelmezett *TDBEdit* helyett).

Egy létrehozott adatszótárt több Delphi alkalmazás is használhat, így a beállításokat csak egy helyen, a szótárban kell elvégeznünk. Van az adatszótáraknak egy másik előnyük is: az adatszótár segítségével az adatbázisból beimportálhatjuk alkalmazásainkba a különböző mezőszintű ellenőrzéseket, így ezek már a kliens oldalon lefutnak: ha rossz jegy értéket ütöttek be, akkor ezt már a kliens program visszadobja, nem kell a rossz értéknek az adatbázis-szerverig elutaznia, hogy majd az utasítsa vissza. Erről a következő pontban még lesz szó.

14.3.3 Az adatszótár létrehozása

Hozzunk létre egy adatszótárt a *Jegyek* adatbázis számára!

Indítsuk el a Delphit. Az új, egyelőre üres alkalmazásunkban hozzunk létre egy új adatmodult. A különböző tábla, lekérdezés... komponensek most nem közvetlenül az álnévre fognak hivatkozni, hanem egy *TDatabase* komponensre. Ennek segítségével az adatbázis és az alkalmazás kapcsolatát globálisan, egységesen kezelhetjük.

---

[2] Az adatszótárak létrehozását és használatát csak a Delphi 32 bites verziói támogatják.

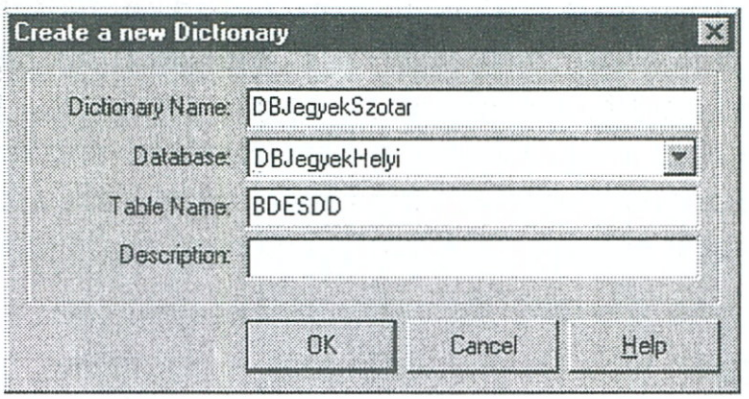
Helyezzünk el az adatmodulon egy *TDatabase* komponenst. Adatait a következőképpen állítsuk be:

```
AliasName = DBJegyek
DatabaseName = DBJegyekHelyi
Name = DBJegyekHelyi
Connected = True
```

Az *AliasName* jellemzőbe a már létrehozott álnevet (*DBJegyek*) állítjuk be. A *DataBaseName*-ben megadott név lesz a helyi, alkalmazásszintű álnév. A leendő *TTable*, *TQuery*, *TStoredProc*... komponensek erre fognak hivatkozni. A *Connected* jellemző Igazra állításával alkalmazásunk hozzákapcsolódott a *Jegyek* adatbázishoz.

Most indítsuk el a Database Explorer segédprogramot. Végezzük el a következő lépéseket:

1. Kattintsunk a *Dictionary* fülre. Itt láthatók a már létező adatszótárak.
2. Hívjuk meg a *Dictionary/New* menüpontot, hogy saját adatszótárunkat létrehozhassuk. A megjelenő párbeszédablakban új adatszótárunk nevét és álnevét kell beállítanunk (14.3. ábra).

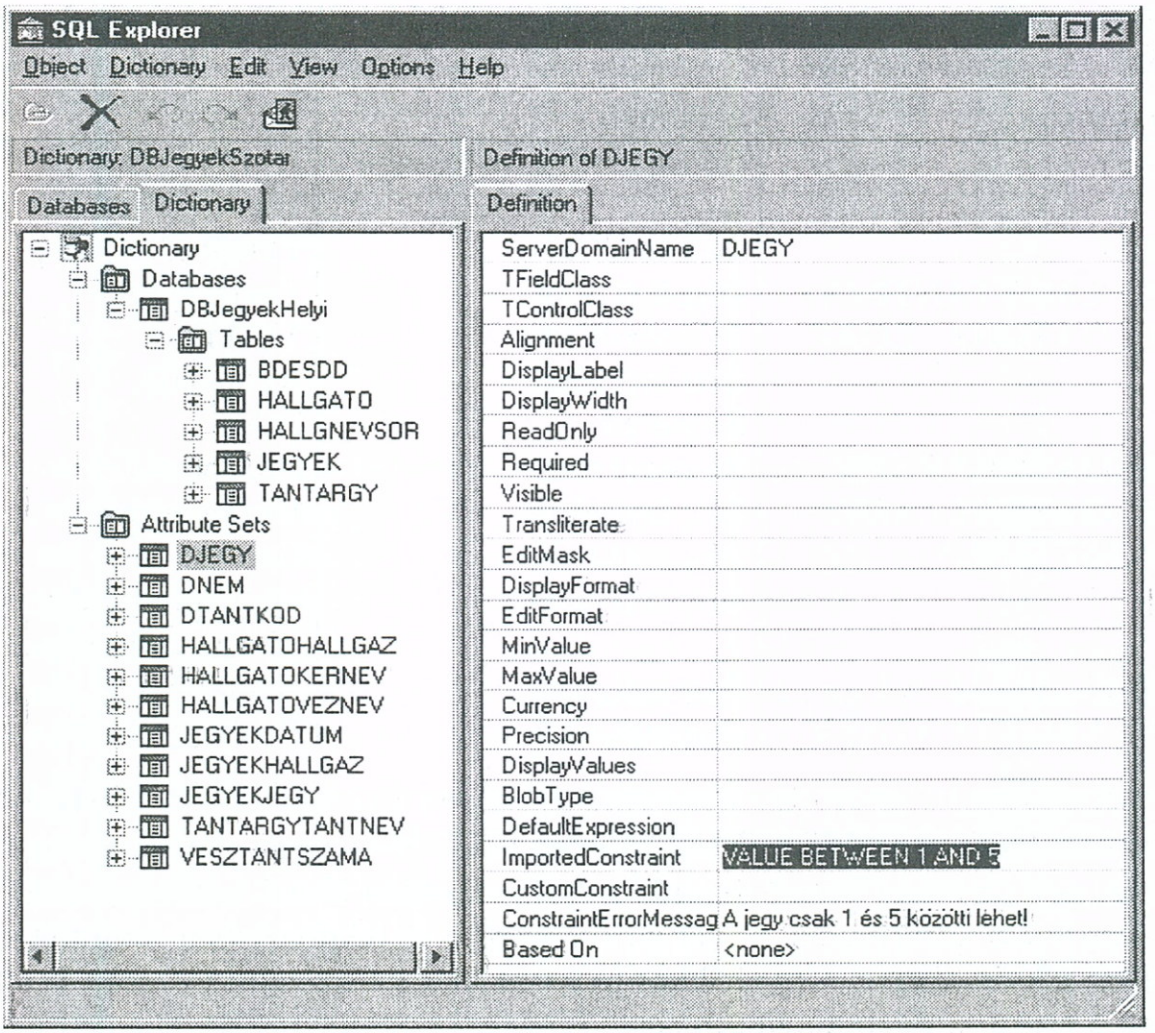


14.3. ábra. Új adatszótár létrehozása

Használhatnánk elvileg a globális *DBJegyek* álnevet is, de akkor alkalmazásunkban kicsit kényelmetlenebbül használhatnánk az adatszótárt.

A BDESDD nevű táblában a rendszer az adatszótár adatait tárolja.

3. Az új adatszótár egyelőre még üres. Hívjuk meg a *Dictionary/Import from Database...* menüpontot, és válasszuk ki a *DBJegyekHelyi* álnevet. Kisvártatva megjelennek a beolvasott adatszótár elemei. Láthatók benne – táblák és az ún. attribútumok formájában – a különböző mezőszintű megszorítások, beleértve a típusokat (*domains*) is (14.4. ábra).



14.4. ábra. A Jegyek adatbázis adatszótára

Azt tapasztaljuk, hogy a *DJEGY* típus alapján létrejött a *DJEGY* attribútum, melyben *Imported Constraint* formájában beolvastuk a jegyre vonatkozó megszorítást. Ez azt jelenti, hogy a rossz jegyértékeket már a Delphi alkalmazásunk ki fogja szűrni, ezek nem fognak a szerverig eljutni. Az ilyenkor megjelenítendő hibaüzenetet írjuk be a *ConstraintErrorMessage* sorba.

Az attribútumok tulajdonságai között felismerni véljük a Delphibeli *TField* mezőobjektumok jellemzőit. A *DisplayLabel*, *EditMask*, *EditFormat*... jellemzőknek azonos az értelmezésük. Ha viszont itt, mármint az adatszótárban állítjuk be az értékeiket, akkor minden további alkalmazásunk – mely ezt az adatszótárt használja – ebből fogja ezeket átvenni, azaz alkalmazásainkban már nem kell egyenként állítgatnunk a mezőobjektumok tulajdonságait. Sőt a *TControlClass* jellemző segítségével egy mezőtípusnak megfeleltethetünk egy Delphi komponenst. Ha például a *DNEM* attribútum *TControlClass* jellemzőjébe a *TRadioGroup*-ot írjuk, akkor a *Hallgato* tábla *Nem* mezőjének űrlapra történő vonszolásakor, a Delphi egy választógomb-csoportot hoz létre. A mezőt a mezőszerkesztőből rávonszoljuk az űrlapra, melyen meg szeretnénk jeleníteni, és ekkor a rendszer a mező típusának megfeleltetett adatmegjelenítési komponenst fogja létrehozni.

4. Vegyük sorba az attribútumokat, állítsuk be tulajdonságaikat a következő táblázat alapján:

Attribútum	Jellemző	Érték
DJEGY	ConstraintErrorMessage	A jegy csak 1 és 5 közötti lehet!
DNEM	TControlClass	TDBRadioGroup
TANTARGYTANTKOD	DisplayLabel	Tantárgykód
HALLGATOHALLGAZ	TControlClass	TDBText
	DisplayLabel	Azonosító
	Required	False[3]
HALLGATOKERNEV	DisplayLabel	Keresztnév
HALLGATOVEZNEV	DisplayLabel	Vezetéknév
JEGYEKDATUM	DisplayLabel	Dátum
	DisplayFormat	yyyy.mm.dd *{egyébként az időt is mutatja}*
JEGYEKHALLGAZ	DisplayLabel	Azonosító
TANTARGYTANTNEV	DisplayLabel	Tantárgynév

A *TControlClass* tulajdonságot csak azoknál az attribútumoknál kell megadnunk, ahol a *TDBEdit*-től eltérő komponenst szeretnénk használni.

Már csak egy „kritikus" mező maradt: a *Hallgato* tábla *VeszTantSzama* mezője. Ennek nincs megfelelő attribútuma, mivel erre a mezőre az adatbázisban semmiféle megszorítást nem alkalmaztunk. Ugyanakkor jó lenne már itt, az adatszótárban beállítani azt, hogy ez a mező minden alkalmazásban *TDBText* komponensben jelenjen meg. Ennek érdekében hozzunk létre számára egy új attribútumot:

5. Hívjuk meg az *Object/New...* menüpontot. Az új attribútum nevét állítsuk VESZTANTSZAMA-ra. Jellemzőit a következőképpen módosítsuk:

Attribútum	Jellemző	Érték
VESZTANTSZAMA	TControlClass	TDBText
	DisplayLabel	Veszélyes tantárgyak száma

6. Mentsük le az adatszótáron végzett beállításainkat. Ennek érdekében jelöljük ki *Dictionary* sort, majd hívjuk meg az *Object/Apply* menüpontot. Ha ezzel végeztünk, zárjuk be a Database Explorer-t, és lépjünk vissza a Delphi alkalmazásunkba.

14.3.4 Az adatmodul felépítése

A kliens/szerver architektúra előnyei a *TQuery* komponensekkel használhatók igazán ki, mivel ekkor mi adhatjuk meg az adatbázis-szerverhez eljuttatandó SQL utasítást, és nem a BDE-nek kell ezt előállítania (az általa előállított utasítássorozat nem lenne mindig optimális). Emiatt **a kliens-szerver architektúrában javasolt a *TQuery* komponensek használata.**

[3] E lépés szükségességéről később szólunk.

Űrlapunkon meg szeretnénk jeleníteni a hallgatókat, valamint az aktuális hallgató jegyeit. Helyezzünk el tehát az adatmodulon két *TQuery* és két *TDataSource* komponenst. Figyelem! A lekérdezések *DatabaseName* jellemzőjét a *TDatabase* komponens által bevezetett helyi álnévre, azaz *DBJegyekHelyi*-re állítsuk.

```
{qryHallgato.SQL}
SELECT * FROM Hallgato

{qryHallgJegyei.SQL}
SELECT * FROM Jegyek
WHERE HallgAz = :HallgAz
```

Állítsuk igazra *TQuery* komponenseink *RequestLive* jellemzőjét. Alapban ez hamis értékű, azaz a lekérdezés eredménye csak olvasható. Így viszont szerkeszthető is lesz, természetesen csak akkor, ha ez az SQL-92 szabvány előírásai szerint is lehetséges (pl. a lekérdezés egy táblán alapul, nem tartalmaz csoportosításokat...).

Minden lekérdezésnél töltsük be a perzisztens mezőket. Figyeljük meg, hogy az adatszótárban beállított jellemzőértékek itt is jelen vannak. Az adatbázisban definiált megszorítások az *Imported Constraint* jellemzőbe kerültek.

Az adatmodul létrehozásakor (az alkalmazás indulásánál) rá kell kapcsolódnunk a szerverre (ezt az adatbázis-komponens *Open* metódusával tesszük). A lekérdezéseket majd az űrlap megjelenítésekor fogjuk megnyitni. Íme a kód:

```
procedure TDM.Bejelentkezes;
begin
  Try
    DBJegyekHelyi.Open;
  Except
    ShowMessage('Érvénytelen felhasználónév vagy jelszó!'+#13#10+
                'A bejelentkezés nem sikerült.');
  End;
end;

procedure TDM.DMCreate(Sender: TObject);
begin
  Bejelentkezes;
end;

procedure TDM.DMDestroy(Sender: TObject);
begin
  DBJegyekHelyi.Close;
end;

{ A qryHallgJegyei lekérdezésnek csak az aktuális hallgató jegyeit
kell tartalmaznia=>a dsrHallgato.OnDataChange-be írjuk a következőket}
procedure TDM.dsrHallgatoDataChange(Sender:TObject; Field:TField);
begin
```

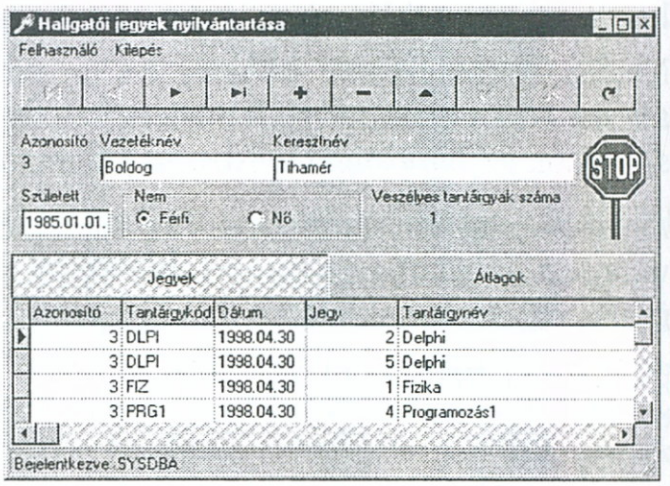
```
  If frmFo <> Nil Then
    begin
      with DM.qryHallgJegyei do
        begin
          Close;
          Params[0].AsInteger:= qryHallgatoHALLGAZ.AsInteger;
          Open;
        end;
    end;
end;
```

14.3.5 Az alkalmazás űrlapjának megtervezése

A feladatspecifikációban leírtak egyetlen fő-segéd űrlapon megvalósíthatók, így ez lesz alkalmazásunk egyetlen ablaka (14.5. ábra).

Az űrlap felső részében a hallgatók adatai láthatók. A navigátorsorral lépegethetünk a hallgatók között. A stoplámpa a bukás veszélyére figyelmeztet. Az űrlap alsó felében a *Jegyek* és az *Átlag* gombok ki-be kapcsolásával az aktuális hallgató jegyei és tantárgyi átlagai jeleníthetők meg.

A *Felhasználó* menüpont az adatbázisba való be- és kijelentkezést teszi lehetővé, a *Kilépés* hatására pedig vége az alkalmazásnak. Az érdeklődő Olvasók a menüpontok implementációját az alkalmazás kódjában találják meg. Ezeket itt nem tárgyaljuk.



14.5. ábra. Alkalmazásunk főűrlapja

Tervezzük meg az űrlapot!

Hívjuk elő a *qryHallgato* mezőszerkesztőjét, jelöljük ki mezőit, majd vonszoljuk ezeket az űrlapra. Azt tapasztaljuk, hogy minden mező számára a rendszer létrehoz egy-egy címkét a mezőobjektum *DisplayLabel*-ben megadott szövegével, és egy-egy adatmegjelenítési komponenst, melynek típusa egyezik az adatszótárban beállítottal.

A stoplámpát egy *Lampa:TImage* komponenssel jelenítjük meg.

A *Jegyek* és *Átlagok* gomboknak kölcsönösen ki kell zárniuk egymást, tehát ezek *TSpeedButton* komponensek lesznek: *spbtnJegyek* és *spbtnAtlagok*. Ne felejtsük el beállítani *GroupIndex* jellemzőjüket egy nullától különböző, kettőjüknél megegyező értékre (pl 1-re). Állítsuk be a *Jegyek* gomb *Down* jellemzőjét Igazra, hogy kezdetben a jegyeket láthassuk.

A rács tartalma a gombok állapotától függ. Ha a *spbtnJegyek* gomb van benyomva, akkor az aktuális hallgató jegyeit, az *spbtnAtlag* gomb hatására viszont ennek átlagait kell tartalmaznia. A gombok *OnClick* jellemzőjében váltogatni fogjuk a rács adatforrását. Mielőtt azonban ezt lekódolnánk, lépjünk vissza az adatmodulra, és készítsük elő az adatforrást.

- Az aktuális hallgató átlagainak kiszámolására az *EgyHallgatoAtlagai* tárolt eljárást fogjuk használni. Mivel ez egy eredményhalmazzal tér vissza, itt nem használható a *TStoredProc* komponens, *TQuery*-re van szükség. Helyezzünk el az adatmodulon egy lekérdezés komponenst és egy ráirányított adatforrást, majd töltsük ki a lekérdezés *SQL* jellemzőjét a következő utasítással:

```
{qEgyHallgAtlagai.SQL}
SELECT * FROM EgyHallgatoAtlagai(:HallgAz)
```

  Állítsuk be a lekérdezés paraméterének típusát Integer-re. A lekérdezést minden egyes *spbtnAtlag* gomb lenyomására újra le kell futtatnunk, hiszen előfordulhat, hogy közben változtak a jegyek.

Az adatforrást elkészítettük, lépjünk vissza az űrlapra, és kódoljuk le a *Jegyek* és *Átlag* gombokat.

```
procedure TfrmFo.spbtnJegyekClick(Sender: TObject);
begin
  Racs.DataSource:= DM.dsrHallgJegyei;
end;

procedure TfrmFo.spbtnAtlagokClick(Sender: TObject);
begin
  With DM.qEgyHallgAtlagai Do
    begin
      Close;
      Params[0].AsInteger:=DM.tblHallgatoHallgAz.AsInteger;
      Open;
    end;
  Racs.DataSource:= DM.dsrqEgyHallgAtlagai;
end;
```

Egy új jegy felvitelénél, módosításánál vagy törlésénél lefutnak az adatbázisban megírt triggerek, melyek újraszámolják az aktuális hallgató veszélyes tantárgyainak számát. Tehát a *Hallgato* táblában a *VeszTantSzama* mező tartalma elvileg megváltozhat. Annak érdekében, hogy ezt a képernyőn is láthassuk, a *qryHallgato* lekérdezést újra le kell futtatnunk. Írjuk be a következő kódot a *qryHallgJegyei* komponens *AfterDelete* és *AfterPost* eseményjellemzőibe:

```
procedure TDM.qryHallgJegyeiAfterDeletePost(DataSet: TDataSet);
begin
  qryHallgato.Close;
  qryHallgato.Open;
end;
```

Számláló típusú mezők a Delphiben

Térjünk vissza a hallgatókhoz. Hogyan veszünk fel egy új hallgatót? Begépeljük adatait, kivéve az azonosítót és *VeszTantSzama* mezőt, majd az egész rekordot elpostázzuk a navi-

gátorsor „pipás" gombjával. Ha a *HallgAz* mező *Required* jellemzőjét nem állítottuk volna az adatszótárban hamisra, akkor már a Delphi alkalmazásunk visszaszólt volna, hiányolta volna az értékét. Ez azért van, mert a Delphi-nek nincs tudomása arról, hogy ez logikailag egy számláló típusú mező, azaz, hogy közvetlenül az adatbázisba való lementés előtt a szerver egy értéket fog számára generálni.
Ezt a problémát két lépésben oldjuk meg:

- Először a *HallgAz* mező *Required* jellemzőjét hamisra kell állítanunk. Ennek hatására a Delphi alkalmazás átengedi, továbbküldi a szerver felé az új rekordot. Az adatbázisban történő tárolás előtt (*Before Insert*) lefut az azonosítónak értéket adó trigger, így a frissen kiosztott azonosító kerül tárolásra.
- Kliens oldali alkalmazásunkban csak akkor fogjuk ezt az értéket látni, ha a *qryHallgato AfterPost* eseményjellemzőjében újraolvassuk az adatokat.

```
procedure TDM.qryHallgatoAfterPost(DataSet: TDataSet);
begin
  with qryHallgato do
     begin
        Close;
        Open;
        Last; //a frissen felvitt rekordra állunk
     end;
end;
```

És most nézzük a bukások figyelését és jelzését. Minden egyes hallgatónál meg kell vizsgálnunk a *VeszTantSzama* mező értékét. Ha ez 0-nál nagyobb, akkor meg kell jelenítenünk a stoplámpát; ellenkező esetben pedig el kell rejtenünk. Ezt a vizsgálatot a *dsrHallgato OnDataChange* eseményébe fogjuk beépíteni (a már meglévő kód mögé):

```
procedure TDM.dsrHallgatoDataChange(Sender: TObject; Field: TField);
begin
  If frmFo <> Nil Then
    ...
    If qryHallgatoVeszTantSzama.AsInteger > 0 Then
      frmFo.Lampa.Visible:= True
    Else frmFo.Lampa.Visible:= False;
end;
```

Ez a kódrészlet az adatmodul egységében található, ezért minősítenünk kell az űrlap nevével (*frmFo.Lampa*). Az űrlap létezésének vizsgálata az alkalmazás indításakor szükséges, mivel az adatmodul még az űrlap előtt megszületik. Ha ezt nem vizsgálnánk, akkor ezzel *Access violation* hibát okoznánk.

Tanulmányozza át a 14_JEGYEK\JEGYEK.DPR alkalmazást. Ebben további érdekes implementációs részletet találhat (pl. a felhasználó nevét és jelszavát bekérő ablak saját tervezésű).
Valósítsa mindezeket meg saját alkalmazásában is!

III. rész
Ínyencségek

E rész tanulmányozásához szükséges előismeretek

Ebbe a részbe kerültek azok a technikát, melyek az előző kettőbe nem illettek. Ezért ennek tartalma elég heterogén. Vannak egyszerűbben, és vannak nehezebben elsajátítható fejezetek is, de mindegyikük érdekfeszítőnek ígérkezik. Például a 16. fejezetben, ahol a súgók készítésével ismerkedhetünk meg, szövegszerkesztői ismereteinkre támaszkodunk. A 19. fejezetben adatbázisos témánk van, a 20.-ban pedig objektumorientált fogalmakról is szó esik.

Sok sikert kívánok e rész elsajátításához!

Mit nyújt ez a rész Önnek?

- **15. fejezet: A komponensek fejlesztése**

  Ebben a fejezetben példákon keresztül megismerkedhetünk a saját komponensek fejlesztésének tág lehetősségeivel.

- **16. fejezet: A súgó készítése**

  Bemutatjuk a windowsos súgóállományok készítésének lépéseit, és ezek használatát a Delphi alkalmazásainkban. A fejezet végén hasznos tippeket és tanácsokat olvashatunk.

- **17. fejezet: A Delphi alkalmazások telepítése**

  Megismerkedhetünk az InstallShield Express program segítségével történő telepítőkészletek létrehozásával.

- **18. fejezet: Az alkalmazások közötti kommunikáció**

  Ebben a fejezetben elsajátíthatjuk az alkalmazások közötti kommunikációnak fortélyait a vágólap, a DDE és az OLE technikák segítségével. Beszélünk a hálózatos DDE kapcsolatról (*NetDDE*), valamint bemutatunk egy OLE automatizmus példát is.

- **19. fejezet: Több rétegű alkalmazások**

  Ebben a fejezetben megismerkedünk az egyre nagyobb teret hódító több rétegű alkalmazások architektúrájával, és Delphibeli létrehozásuknak módjával egy konkrét három rétegű adatbázisos feladaton keresztül.

- **20. fejezet: Több szálon futó alkalmazások**

  Megismerkedünk a szálak (*threads*) fogalmával, majd készítünk egy konkrét adatbázisos két szálon futó alkalmazást. Ebben arra is fény derül, hogy az adatfeldolgozásokban az SQL lekérdezések gyorsabban szolgáltatnak eredményt, mint a programból történő feldolgozás.

15. A komponensek fejlesztése

A Delphi 3.0 kliens/szerver környezetben kb. 150 komponens található a különböző komponenspalettákon. Ezek tulajdonképpen Object Pascalban fejlesztett osztályok (közvetlenül vagy közvetve mind a *TComponent* osztály leszármazottjai), melyek az ún. regisztrálási folyamatnak köszönhetően felkerültek a megfelelő komponenspalettákra. A komponensek forrásait a DELPHI\SOURCE\VCL (Visual Component Library) könyvtárban találhatjuk[1].

Mi is készíthetünk ilyen komponenseket, és erre a konkrét feladatokban szükség is lehet. Ha például egy felhasználó azt kéri, hogy a szerkesztődobozokból az *Enter* billentyű hatására is lépjünk ki (mint a *Tab*-ra), akkor kérését legkényelmesebben egy új, speciális szerkesztődoboz komponens fejlesztésével és felhasználásával elégíthetjük ki. Ez egy olyan szerkesztődoboz lesz (nyilván a *TEdit*-ből fog származni), mely az *Enter* billentyű hatására átadja a fókuszt a soron következő vezérlőelemnek. De természetesen írhatunk más komponenseket is: például egy olyan szerkesztődobozt, melynél már tervezési időben beállítható az elfogadható karakterek halmaza (csak kisbetűket, csak számjegyeket... fogadjon el), vagy készíthetünk egy olyan listadobozt, mely *OnScroll* eseményjellemzővel is rendelkezik... A lehetőségek korlátlanok. Erről úgy is meggyőződhetünk, hogy „bóklászunk" egy kicsit Interneten vagy a Compuserve hálózatán. Különböző programozók a nagyvilágból egymással versenyezve kínálják komponenseiket.

Egy új komponens fejlesztése tehát egy új osztály létrehozásából, majd ennek regisztrálásából áll. A hangsúly az új osztály megtervezésén és kivitelezésén van: vannak egyszerűbb komponensek, melyek csak kicsit különböznek a már létezőktől, és vannak bonyolultabb komponensek, melyeknél az írandó programsorok száma több ezer is lehet. Itt a nehézséget leginkább a számtalan örökölt adat, jellemző és metódus közötti összefüggés átlátása jelenti, főleg amikor új osztályunknak több mint 10 ősosztálya van. Ha a létező komponenseken alapulva hozzuk létre saját komponensünket, akkor az sem árt, ha jól tudunk angolul. És mindehhez természetesen teljes objektumorientált programozási tudáskészletünket is be kell vetnünk. Ezért vallják sokan, hogy Delphiben az igazi kihívást – és a programozói megelégedettséget is – az új komponensek fejlesztése jelenti.

Ebben a fejezetben bepillantást nyerünk a komponensek fejlesztésének világába. Egyszerű példákon keresztül megtudhatjuk, hogyan kell új komponenseket létrehozni és regisztrálni, hogyan lehet az új komponens ikonját megrajzolni és beállítani, valamint azt is, hogy hogyan csoportosíthatók ezek komponenskönyvtárakba.

---

[1] A komponensek forrása csak a Delphi Developer és Client/Server változataihoz jár.

A következő komponenseket fogjuk elkészíteni:

- *TAlignButton*: olyan gomb, melynek *Align* jellemzőjével már tervezési időben beállíthatjuk az igazítását: *Align=alTop* ⇨ a szülőkomponens (általában űrlap) tetejéhez ragad...
- *TIncCombo*: olyan kombinált listadoboz, melybe részkarakterláncra való rákeresést (*incremental search*) építünk be, vagyis minden egyes újonnan leütött betűnél rákeresünk a már begépelt szöveggel kezdődő listaelemekre, és találat esetén begépelt szövegünket kiegészítjük a talált szóval.
- *TEnabEdit*: olyan szerkesztődoboz, melynél tervezési időben beállítható az elfogadható karakterek halmaza.
- *TScrollList*: olyan listadoboz, melynek *OnScroll* eseményjellemzője is van.
- *TAboutBox*: egy névjegyablak, melyet ugyanúgy tudunk majd használni, mint a rendszer-párbeszédablakokat: elhelyezzük egy űrlapon, beállítjuk az objektum-felügyelőben az adatait (hogy mi a program neve, ki írta...), majd az *Execute* metódussal megjelenítjük.

15.1 A komponensfejlesztés lehetőségei

Legtöbbször az új komponenst valamely létező, *TComponent*ből származó osztály utódjaként hozzuk létre: egy kicsit speciálisabb gomb a *TButton*-ból származna, a speciális szerkesztődoboz a *TEdit*-ből... Az ősosztályt természetesen a feladattól függően választjuk ki. Ilyenkor az utód osztályt a következőképpen specializálhatjuk:

- Egy örökölt jellemző láthatóságának megváltoztatása

  Például egy nyilvános jellemzőt tervezési időben is elérhetővé tehetünk úgy, hogy az utód osztályban publikálttá deklaráljuk. Így fogjuk a *TAlignButton* komponenst létrehozni, lásd a 15.2. pontban.

  Figyelem, a láthatóságot nem lehet szűkíteni, csak bővíteni!

- Már meglévő metódusok és eseménykezelők felülírása

  Például ha meg szeretnénk szűrni a szerkesztődobozba írható karaktereket, akkor a *KeyPress* metódust kell átírnunk (lásd *TEnabEdit, 15.6.* pont).

- Új mezők, jellemzők és metódusok hozzáadása

  A *TEnabEdit* komponensnél szükségünk lesz egy új jellemzőre, melyben tervezéskor megadhatjuk az elfogadható karaktereket.

- Új eseménykezelők hozzáadása

  A *TScrollListBox* komponensnél egy új eseménykezelőt kell bevezetnünk, az *OnScroll*-t (15.7. pont).

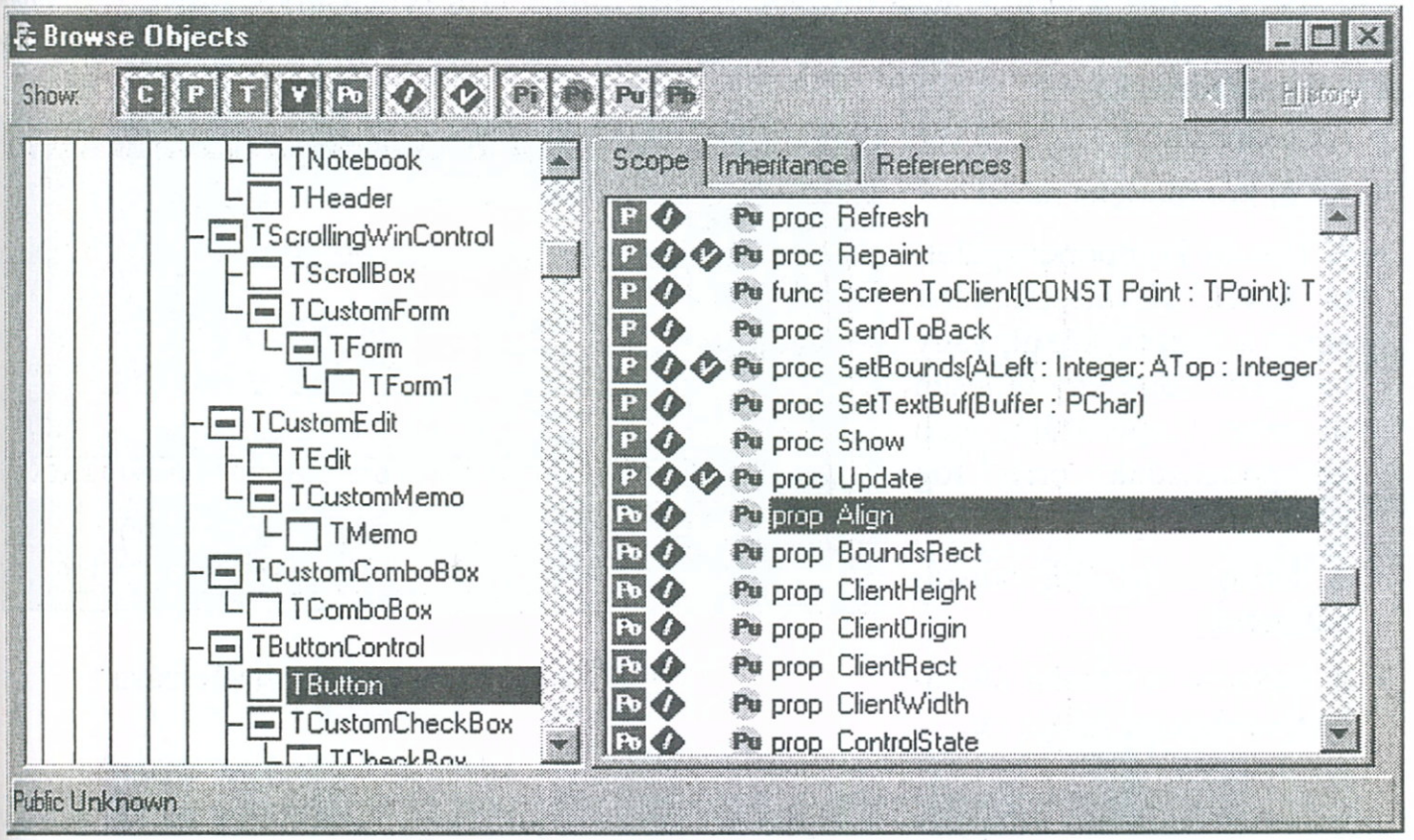
Legtöbbször egy új komponensnél ezen specializálási módszerek mindegyikét alkalmazzuk.

15.2 TAlignButton

Készítsünk egy olyan gombkomponenst, melynél be lehet állítani a szülőkomponensen belüli igazítást. Rendelkezzen tehát egy *Align* jellemzővel: ha *Align=alTop*, akkor a gomb igazodjon az űrlap tetejéhez, ha *Align=alBottom*, akkor az aljához...

Megoldás (🖫 15_KOMPONENSEK\ALIGNBUTTON.PAS)

Bármennyire is furcsálljuk, a beépített *TButton* komponenst nem lehet igazítani. Az igaz, hogy nagyon sok komponensnél jelen van az *Align* jellemző, de a *TButton*-nál nincs. A helyzet megvizsgálására jelenítsük meg a beépített osztályhierarchiát. A *View/Browser* menüpont az aktuális alkalmazásba befordított komponensek osztályhierarchiáját jeleníti meg. Éppen emiatt csak akkor aktív, ha az alkalmazásunkat már lefordítottuk. Fordítsunk le egy üres alkalmazást, majd hívjuk meg a menüpontot!

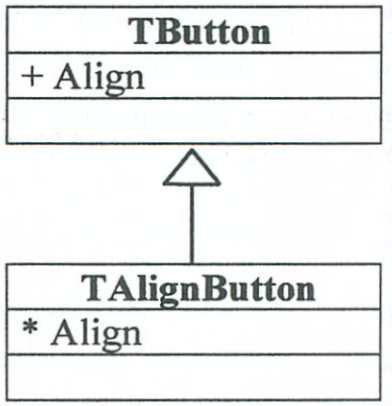


15.1. ábra. A Delphi komponensek osztályhierarchiája

A megjelenő ablaknak két fő része van: a bal listában az osztályhierarchia látható, míg a jobb oldaliban a kijelölt osztály mezői, jellemzői, metódusai. Az űrlap tetején található gombokkal beállíthatjuk a megjelenítendő adatokat: ha például benyomjuk a *Pi* feliratú gombot, akkor a privát adatok és metódusok is megjelennek; ha a *V* feliratút is benyomjuk, akkor a virtuális metódusok is láthatók stb.

Keressünk rá a *TButton* osztályra. Ennek érdekében előbb kattintsunk az űrlap bal részébe, majd gépeljük be a keresett osztály nevét. Miután megtaláltuk, ugyanígy keressük meg a gomb *Align* jellemzőjét az űrlap jobb részében.

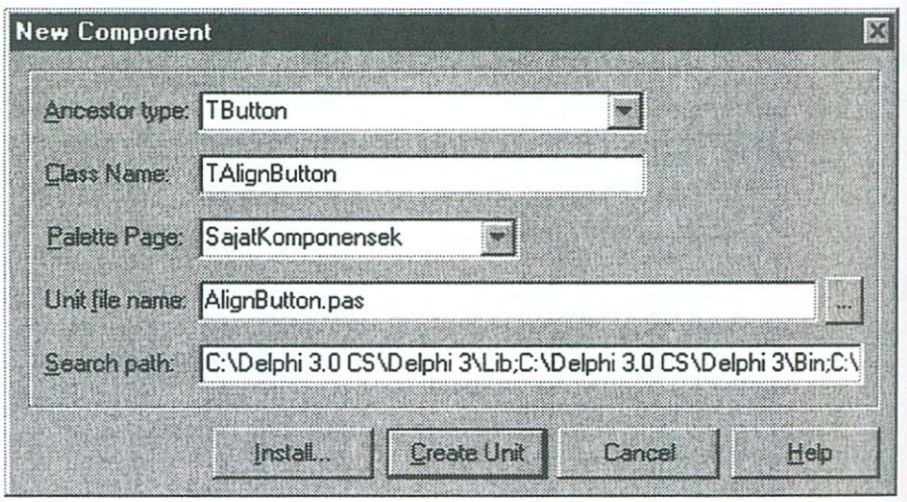
Azt tapasztaljuk, hogy a beépített gombnak tulajdonképpen már van *Align* jellemzője, ez azonban tervezéskor nem állítható, mivel nyilvánosnak lett deklarálva. Hozzunk létre egy új osztályt, melyben ezt a jellemzőt publikálttá tesszük (15.2. ábra).



15.2. ábra. Osztálydiagram[1]

Következzen most a komponens megvalósítása. Ennek érdekében hívjuk meg a *Component/New Component...* menüpontot[2]. A megjelenő űrlapon (15.3. ábra) a következő adatokat kell megadnunk:

- Az őszosztályt
- Az új komponens nevét
- A leendő komponenspaletta nevét. Kiválaszthatunk egyet a létezők közül, vagy begépelhetünk egy teljesen új nevet. Regisztrálás után komponensünk erre fog kerülni.
- A komponens állományának nevét
- A keresési útvonalat általában a rendszer állítja be.



15.3. ábra. Új komponens létrehozása

A komponensvarázsló űrlapjából kilépve azt tapasztaljuk, hogy a rendszer létrehozott egy új egységet (*unit AlignButton*), melyben máris megtalálható az új komponens definíciója. Már csak a lényeget kell begépelnünk.

---

1 Emlékeztetőül: a – privát (*private*); a # védett (*protected*); a + nyilvános (*public*); a * publikált (*published*) láthatóságot jelöl (bővebben lásd a 3. fejezetben).

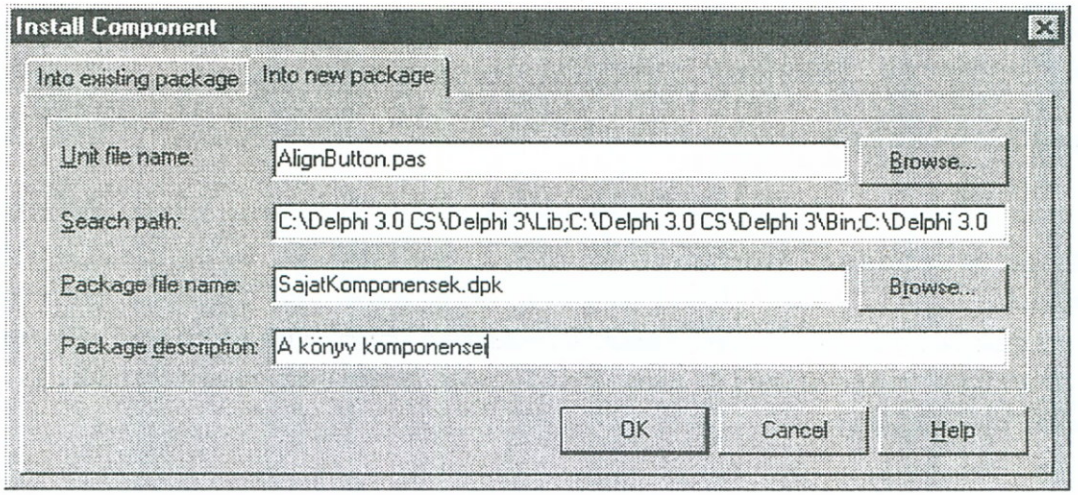
2 Az említett menüpont csak a Delphi 3-ra érvényes. Delphi 1-ben új komponenst a *File/New Component...* menüponttal, míg Delphi 2-ben a *Component/New...*-val hozhatunk létre. A beállítandó adatok nagyjából megegyeznek mindhárom verzióban.

```
unit AlignButton;
interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
     Dialogs, StdCtrls;

type
  TAlignButton = class(TButton)
    published
      {Újradeklaráljuk a jellemzőt, immár publikáltként. Típusát és
      Read/Write metódusait nem kell még egyszer megadnunk.}
      property Align;
  end;

procedure Register;

implementation

{Ez az eljárás végzi el a komponens regisztrációját.}
procedure Register;
begin
  RegisterComponents('SajatKomponensek', [TAlignButton]);
end;

end.
```

Mentsük le a komponenst, majd következhet a regisztráció (telepítés). Ennek érdekében hajtsuk végre a következő lépéseket:

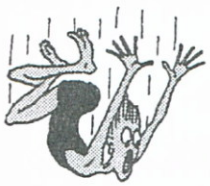
- Hívjuk meg a *Component/Install Component...* menüpontot[1].
- Az *Install Component* párbeszédablakban lépjünk át az *Into new package* fülre, ezzel komponensünket egy új komponenscsomagba fogjuk befordítani (a csomagokkal a következő pontban foglalkozunk). Töltsük ki az adatokat (15.4. ábra):
  - Komponensünk állományának nevét: ALIGNBUTTON.PAS
  - A keresési útvonalat a rendszer kezeli
  - Gépeljük be az új komponenscsomag nevét: *SajatKomponensek*
  - Jellemezzük pár szóban az új csomagot: *A könyv komponensei*
- Zárjuk be az ablakot. Az ezután megjelenő párbeszédablak arra figyelmeztet, hogy az új csomagot a rendszer most fogja lefordítani és telepíteni. Hagyjuk jóvá. Végül tapasztalni fogjuk, hogy komponensünk beépült a meglévők közé egy *Sajat-Komponensek* feliratú palettára. Ikonja egyelőre még megegyezik a *TButton*-éval, de már nem sokáig; a 15.4. pontban egy új ikont fogunk számára rajzolni.

[1] Delphi 1-ben *Options/Install Components...*, Delphi 2-ben *Component/Install...*



15.4. ábra. Egy új komponens telepítése

Ezzel vége a telepítésnek. Próbáljuk ki új komponensünket! Helyezzük el egy példányát az űrlapon, és állítsuk be *Align* jellemzőjét. Mi történik?

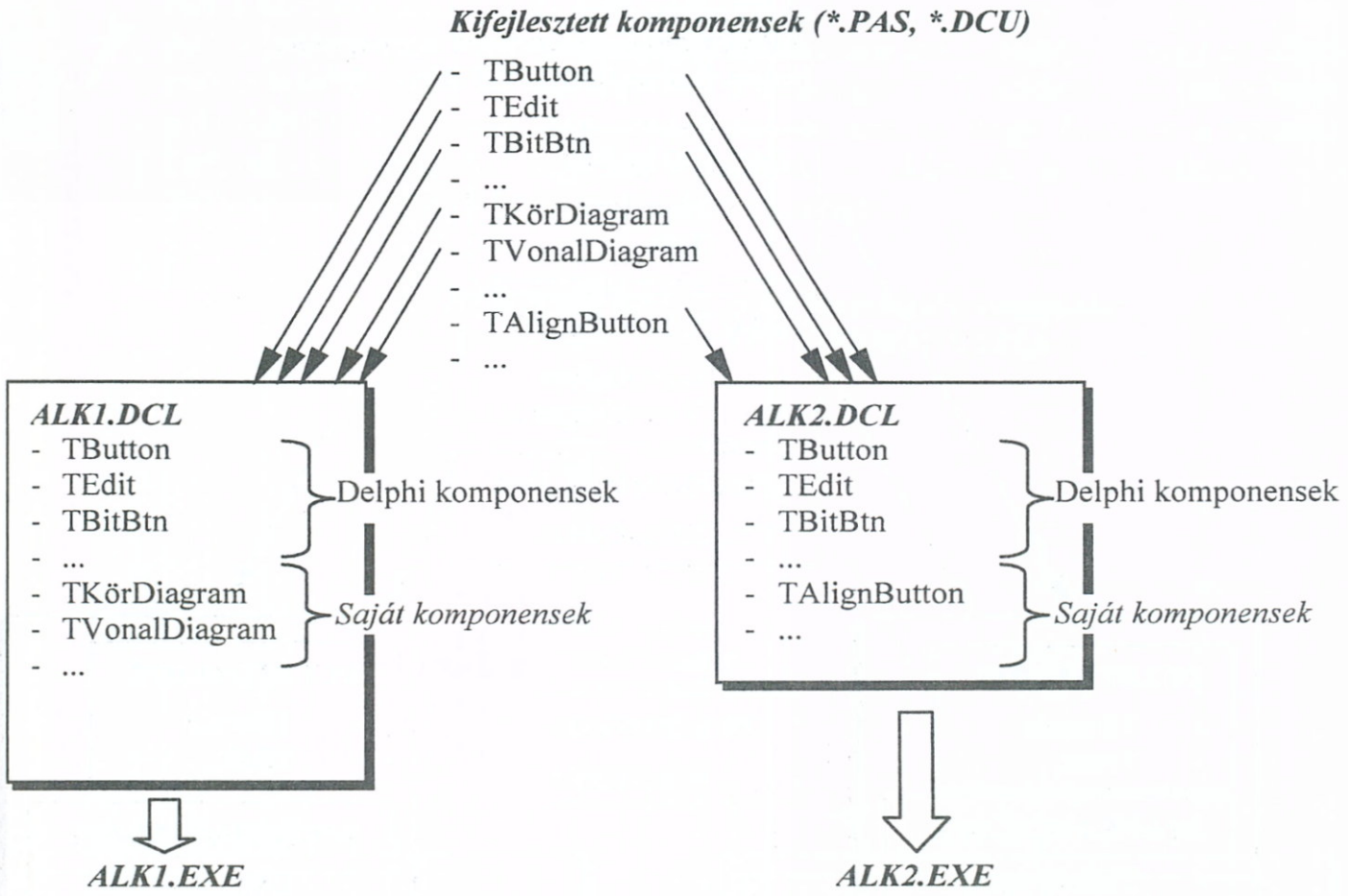


15.3 A komponenscsomagok fogalma

Telepítéskor a komponens a Delphi rendszer részévé válik. Ez azt jelenti, hogy lefordított kódja beépül a rendszer által használt **komponenskönyvtárba *(Component Library)***. **Delphi 1-ben** ennek a könyvtárnak COMPLIB.DCL[1] a neve, míg **Delphi 2-ben** CMPLIB32.DCL (Delphi 3-ban kicsit más a helyzet, erről később szólunk). A rendszer alapértelmezés szerint ezekkel az állományokkal dolgozik. Természetesen létrehozhatunk saját komponenskönyvtárakat is (ALK1.DCL, ALK2.DCL...), és erre a különböző alkalmazások fejlesztésekor szükség is lehet. Általában minden alkalmazás használja a Delphibe eleve beépített komponenseket (vagy ezek nagy részét), és ezen kívül még saját komponenseket is igénybe vehet. Két különböző alkalmazás egyáltalán nem biztos, hogy pontosan ugyanazokat a komponenseket veszi igénybe. Ezért szoktunk létrehozni saját, alkalmazásszinten testre szabott komponenskönyvtárakat. Mindegyik alkalmazás fejlesztésekor ennek saját könyvtárát töltjük be a rendszerbe (Delphi 1-ben: *Options/Open Library*, Delphi 2-ben: *Component/Open Library...* menüponttal). Az alapértelmezés szerinti könyvtárba pedig csak az általánosan használt komponenseket fordítjuk be (ez a COMPLIB.DCL állomány).
A rendszer egyszerre csak egy komponenskönyvtárral dolgozhat, ez mindig az aktuálisan fejlesztendő alkalmazás könyvtára lesz. Ezt tapasztalhatjuk a Delphi 1-es és 2-es változataiban (15.5. ábra).

---

[1] COMPLIB.DCL = **Comp**onent **Lib**rary. **D**elphi **C**ompiled **L**ibrary
Ezek az állományok a DELPHI\BIN könyvtárban találhatók.



15.5. ábra. Mindegyik alkalmazás a saját speciális komponenskönyvtárával

Delphi 3-ban bevezették a **komponenscsomagokat** (***component packages***), melyek még nagyobb rugalmasságot visznek Delphi környezetünkbe. Egy adott pillanatban már nem csak egy komponenskönyvtárral dolgozhatunk, hanem egyszerre akárhánnyal. Nézünk néhányat ezek közül:

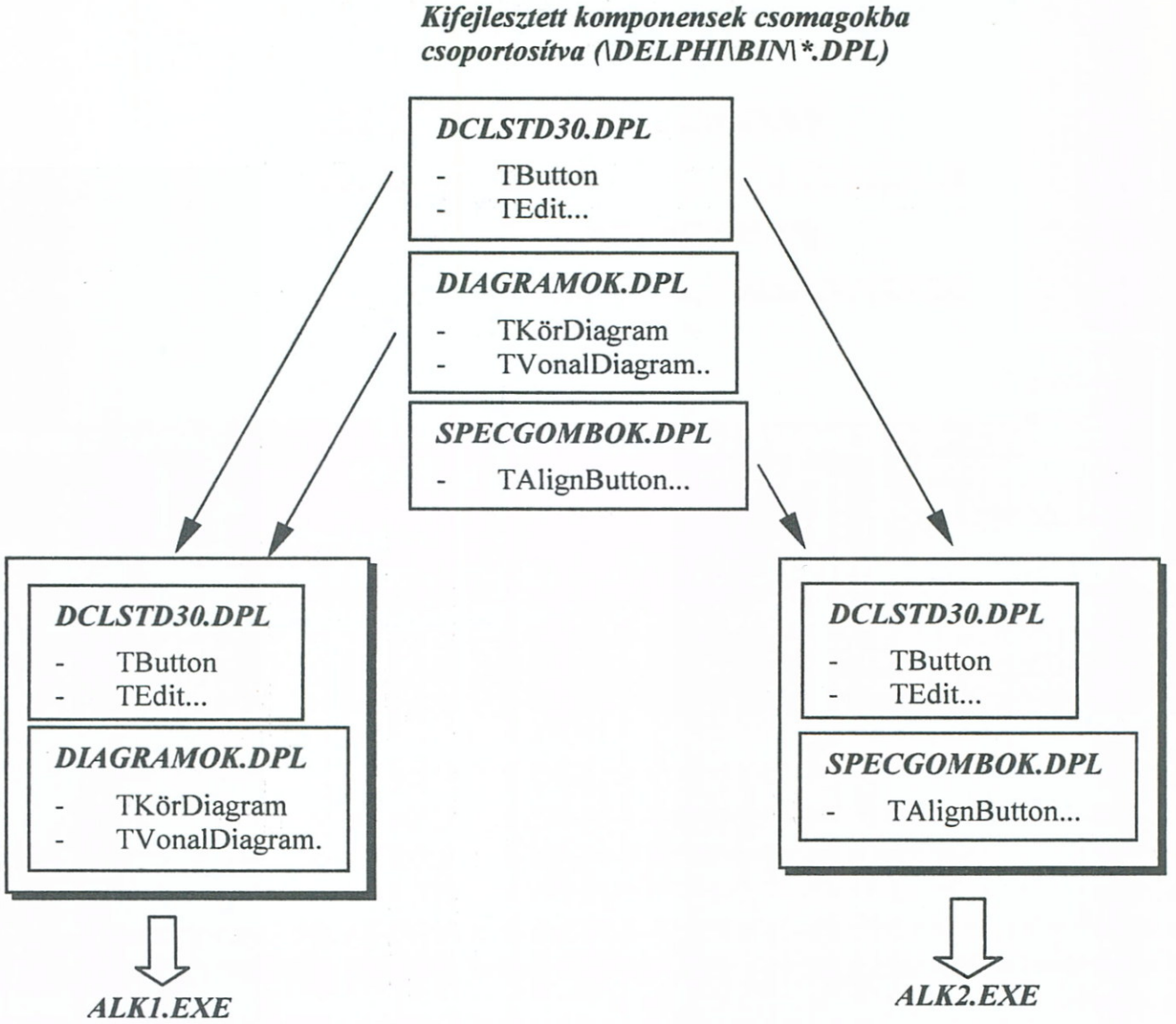
- DCLSTD30.DPL[1] = Delphi standard komponensek csomagja,
- DCLQRT30.DPL = QuickReport komponenscsalád,
- DCLSMP30.DPL = mintakomponensek (*sample,* innen az *SMP* az állomány nevében),
- és még sok más csomag is. Mindezek a DELPHI\BIN könyvtárban találhatók

A felhasználó (*user*) komponenseinek telepítésekor a rendszer felkínálja a DCLUSR30.DPK[2] csomagot. Természetesen ezt nem kötelező igénybe venni, létrehozhatunk saját csomagokat, sőt saját csomagkollekciókat (*.DPC = **D**elphi **P**ackage **C**ollection) is.

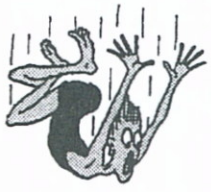
---

[1] DCLSTD30.DPL = **D**elphi **C**omponent **L**ibrary **Std**. **D**elphi **P**ackage **L**ibrary

[2] *.DPK = *. **D**elphi **Pack**age; lefordítása után ebből lesz egy *.DPL és egy *.DCP (**D**elphi **C**ompiled **P**ackage).



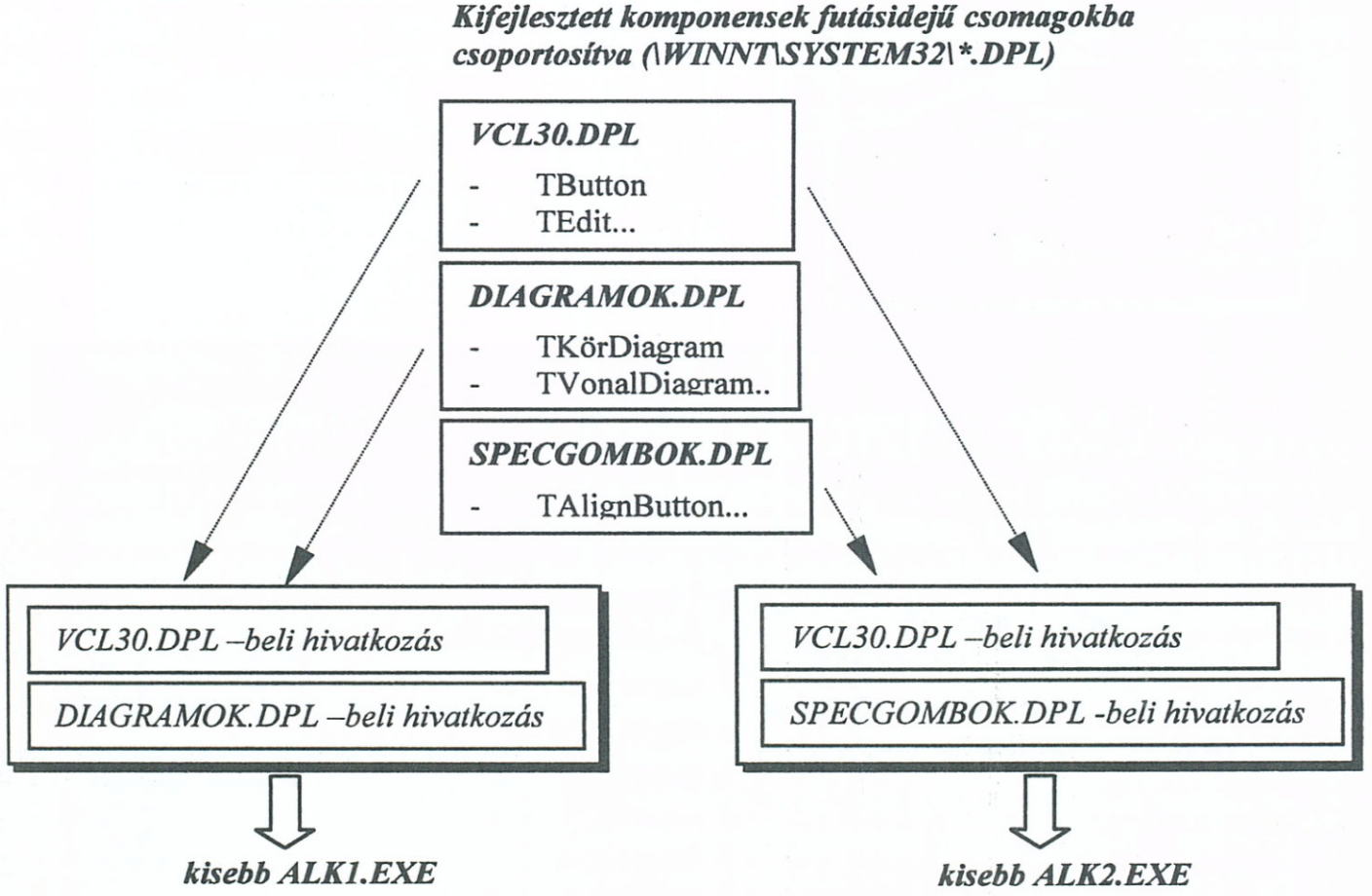
15.6. ábra. Alkalmazások a Delphi 3-ban, ha nem használunk futás idejű csomagokat



Elemezzük a 15.6. ábrát: alkalmazásaink EXE állományába a rendszer befordítja a tervezési időben használatos komponenskönyvtárakat. Ezek a könyvtárak a *Component/Install packages...* menüpont hatására megjelenő ablak felső felében (*Design packages*) tekinthetők meg.

A csomagoknak, a nagyobb rugalmasságon kívül, van egy másik nagy előnyük is. Generálhatunk olyan alkalmazásokat (az előbb említett ablakban a *Build with runtime packages* jelölőnégyzet állapotán múlik csupán), melyek ***futásidejű csomagokat*** használnak. Az így lefordított EXE állomány nem fogja tartalmazni a komponensek lefordított kódját (és emiatt mérete jelentősen csökken, akár 10 KB-ra is), hanem ezt futás közben az ún. futás idejű csomagokból olvassa majd ki. A futásidejű csomag elve nagyon hasonlít a DLL technikáéhoz: több alkalmazás – futás közben – megosztva használ egy könyvtárat, ami-

nek a kiterjesztése itt DPL, nem pedig DLL.
Ahhoz, hogy futás idejű csomagokat használhassunk szükséges, hogy az alkalmazás lefordításakor kéznél legyenek a szükséges DCP állományok (ezek helye a DELPHI\LIB könyvtárban van), valamint arra is, hogy futáskor a használt DPL-ek is elérhetők legyenek (ezeknek – akárcsak a DLL-eknek – a WINNT\SYSTEM32 könyvtárban kell lenniük).



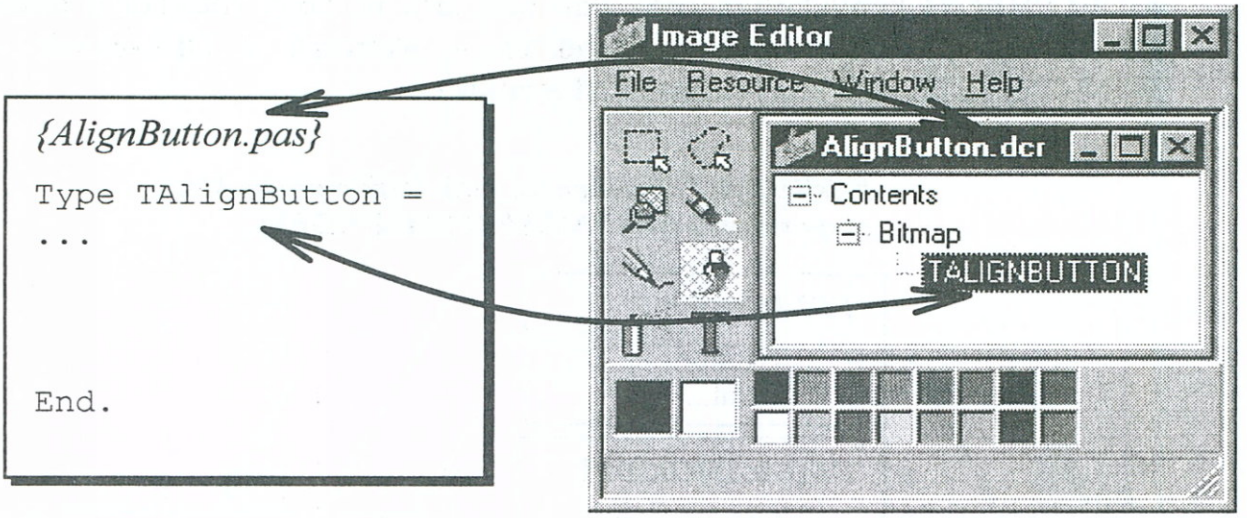
15.7. ábra. Kisebb EXE állományok a futásidejű csomagok segítségével

15.4 Komponens ikonjának beállítása

A komponens képe egy 24*24-es bittérképből származik, melyet az Image Editor segédprogrammal fogunk megrajzolni. A bittérképet tartalmazó állomány kiterjesztése DCR lesz (***Delphi Component Resource***).
Szabályok (15.8. ábra):

- A komponens képét tartalmazó állománynak (DCR) és a komponens leírását tartalmazó egységnek (PAS) ugyanabban a könyvtárban kell lenniük.
- A komponens képét tartalmazó állomány nevének meg kell egyeznie a komponens egységének nevével (ALIGNBUTTON.PAS ⇒ ALIGNBUTTON.DCR)

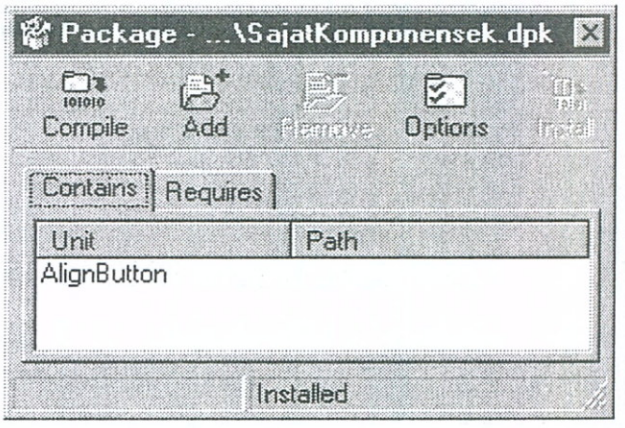
- A bittérkép nevének meg kell egyeznie a komponens típusának nevével (*TAlignButton* a típus ⇒ *TALIGNBUTTON* a bittérkép neve). Figyelem, a kép nevét csupa nagybetűvel írjuk!



15.8. ábra. A komponens leírása és az ikonja közötti összefüggés

Rajzoljuk meg az új gombkomponensünk képét, majd mentsük le az ALIGNBUTTON.DCR állományba. Annak érdekében, hogy a rendszer figyelembe vegye az új rajzot, hajtsuk végre a következő lépéseket[1]:

- Hívjuk meg a *Component/Install Packages...* menüpontot. A megjelenő ablakban a rendszerben jelenlévő komponenscsomagokat láthatjuk (pontosabban a leírásukat).
- Jelöljük ki *A könyv komponensei* csomagot, majd kattintsunk az *Edit* feliratú gombra. Betöltődik a csomagszerkesztő párbeszédablak.
- Jelöljük ki benne az *AlignButton* komponenst, majd töröljük ki (*Remove*) a csomagból.
- Olvassuk be újra komponensünket (*Add*).
- Fordítsuk újra a komponenscsomagot (*Compile*).
- A csomagszerkesztő párbeszédablak bezárása után tapasztalni fogjuk, hogy komponensünk képe megváltozott.



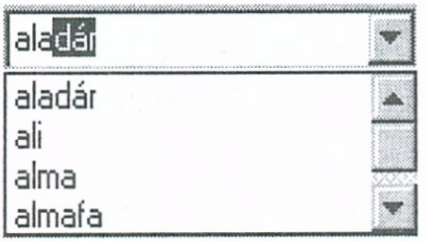
15.9. ábra. A csomagszerkesztő ablak

[1] Az itt leírt lépések a Delphi 3-ra vonatkoznak. Delphi 1-ben az *Options/Rebuild Library* menüponttal, míg a Delphi 2-ben a *Component/Rebuild Library*-val lehet újrafordítani a komponenskönyvtárat, és ezzel érvényesíteni a képet.

Ha a gomb ikonja nem változott meg, akkor még egyszer ellenőrizzük le figyelmesen a DCR állományra és a bittérképre vonatkozó követelményeket.

15.5 TIncCombo

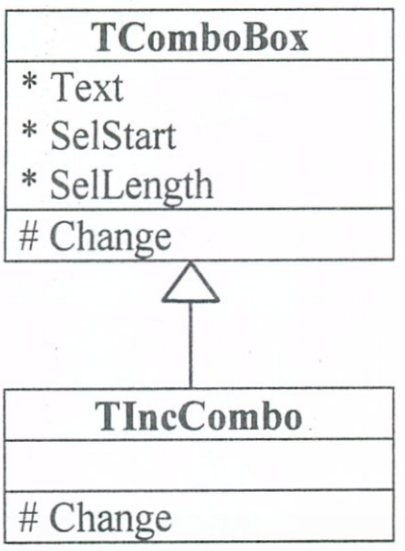
Készítsünk egy olyan listadobozt, melyben az éppen begépelt részszövegre a gép automatikusan rákeres. Ha a listában van egy olyan elem, melynek első betűi megegyeznek a begépelt betűkkel, akkor egészítse ki a szöveget a szerkesztődobozban. A kiegészítés legyen kijelölve annak érdekében, hogy egy további betű leütésekor ez automatikusan tűnjön el.



Megoldás (15_KOMPONENSEK\INCCOMBO.PAS)

Minden egyes leütött betűnél egy kereső kódrészletnek kell lefutnia. Keressük meg a megfelelő eseményt, amire a kereső kódrészletet beépíthetjük. A szóba jöhető események a következők: *OnKeyDown*, *OnKeyPress*, *OnChange*. Komponensek fejlesztésekor nem magát az eseményjellemzőt szoktuk lekódolni, hiszen ez a komponens felhasználójának lett kitalálva. Minden eseményjellemzőnek van egy párja, egy védett (*protected*) virtuális metódus: *OnKeyPress* ⇨ *KeyPress*, *OnChange* ⇨ *Change* stb. Ezeket a metódusokat fogjuk a komponensek fejlesztésekor felülírni, megváltoztatni.

Vizsgáljuk meg a szóba jöhető metódusokat: a *KeyDown* nem jó, hiszen minden billentyű leütésénél bekövetkezik – még a nyíl billentyűknél is – holott akkor a szöveg nem is változik. A *KeyPress* azért nem használható, mert lefutásának pillanatában a kombinált lista *Text* jellemzője még nem tartalmazza a frissen leütött billentyűt is. A megoldás a *Change* metódus. Ezt kell felülírnunk a *TComboBox* osztályból származó komponensünkben.
Valahányszor változás áll be a kombinált lista szövegében, mindannyiszor elvégezzük a keresést, és találat esetén a *SelStart* és *SelLength* jellemzők segítségével kijelöljük a kiegészített szövegrészt.



15.10. ábra. Osztálydiagram

Nézzük a kódot:

```
unit IncCombo;

interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
     Controls, Forms, Dialogs, StdCtrls;

type
   TIncCombo = class(TComboBox)
      protected
         Procedure Change; override;
   end;

procedure Register;
implementation

procedure TIncCombo.Change;
   function ElejeEgyezik(Minek, Miben:String):Boolean;
      begin
         ElejeEgyezik:= Pos(Minek, Miben)=1;
      end;

   Var i, poz:Integer;
   begin
      i:=0;
      While (i<= Items.Count-1) And Not ElejeEgyezik(Text, Items[i]) Do
         Inc(i);
      If i<= Items.Count-1 Then {Ha találtunk valami jót}
         begin
            {Kimentjük a kurzor pozícióját}
            poz:= Length(Text);
            {Fölvisszük a szöveget a szerkesztődobozba}
            ItemIndex:=i;
            {Kijelöljük a maradékot}
            SelStart:= poz;
            SelLength:= Length(Text)- poz;
         end;
      {Jöjjön, aminek jönnie kell}
      Inherited Change;
   end;

procedure Register;
begin
   RegisterComponents('SajatKomponensek', [TIncCombo]);
end;

end.
```

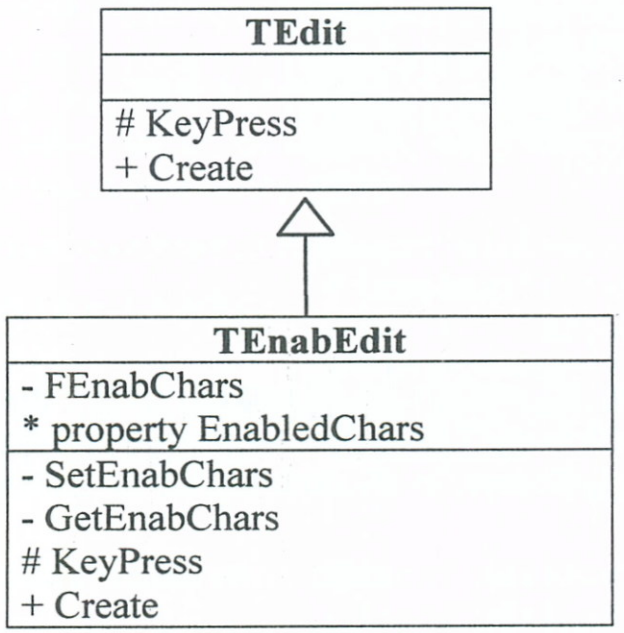
Telepítse, majd tesztelje le az új komponenst! Miért nem lehet visszafele törölni?

15.6 TEnabEdit

Most készítsünk egy olyan szerkesztődoboz komponenst, melynél már tervezési időben beállíthatjuk az elfogadható karaktereket. Ha például beállítjuk, hogy csak az 'abc' karaktereket fogadja el, akkor semmi mást ne lehessen beírni. Valamely tiltott karakter begépelésekor sípoljon egy rövidet.

Megoldás (15_KOMPONENSEK\ENABEDIT.PAS)

Új komponensünk a *TEdit* osztály utódja lesz. Be kell vezetnünk egy új jellemzőt (*property EnabChars*), melynek értékét már tervezéskor is állíthatjuk. Ennek típusa *String* lesz. Háttérében az *FEnabChars* privát mező fog állni; ezt karakterhalmaznak fogjuk deklarálni, hiszen így egy betűről kényelmesebben megállapítható, hogy az elfogadható betűk között van-e (halmaz elemvizsgálattal). A karakterlánc ↔ karakterhalmaz konverziót a jellemző író/ olvasó metódusai fogják elvégezni.
Az új privát mező kezdőértékét nem szabad a véletlenre bíznunk. Felülírjuk a konstruktort, és ebben beállítjuk kezdőértékét. Mivel ezt a komponenst inkább a csak néhány karaktert elfogadó szerkesztődobozokhoz fogjuk használni, az elfogadható karakterek halmaza legyen kezdetben üres.
A tényleges szűrést a *KeyPress* metódusban valósítjuk meg.



15.11. ábra. Osztálydiagram

```
unit EnabEdit;

interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls;

type
  TEnabEdit = class(TEdit)
    private
      FEnabChars: Set of Char;
```

```
      procedure SetEnabChars(Value:String);
      function GetEnabChars:String;
    protected
      procedure KeyPress(var Key:Char);override;
    public
      constructor Create(AOwner:TComponent);override;
    published
      property EnabledChars:String    read GetEnabChars
                                      write SetEnabChars;
  end;

procedure Register;

implementation

constructor TEnabEdit.Create(AOwner:TComponent);
begin
  Inherited Create(AOwner);
  FEnabChars:=[];
end;

procedure TEnabEdit.SetEnabChars(Value:String);
var i:Integer;
begin
  FEnabChars:=[];
  For i:=1 to length(Value) Do
    FEnabChars:=FEnabChars+ [Value[i]];
end;

function TEnabEdit.GetEnabChars:String;
var I:Integer;
begin
  Result:='';
  For I:=0 to 255 Do
    If Chr(I) In FEnabChars Then
      Result:= Result+ Chr(I);
end;

procedure TEnabEdit.KeyPress(var Key:Char);
begin
  If (Key In FEnabChars) or (Key = #8) Then
    {Ha minden OK, akkor történjen, ami szokott,}
    Inherited KeyPress(Key)
  Else
    {egyébként lenullázzuk a karaktert, és ezzel megszakítjuk
    az üzenetláncot}
    begin
```

```
        Key:=#0;
        Beep;
      end;
end;

procedure Register;
begin
  RegisterComponents('SajatKomponensek', [TEnabEdit]);
end;

end.
```

15.7 TScrollList

A 6. fejezet feladatai között volt egy olyan is, amelyben két, egymás melletti listadoboz elemeit vonszolással összekapcsoltuk. A kapcsolatot jelző vonalat minden listagördítéskor újra kellett rajzolnunk. Ekkor szomorúan tapasztaltuk, hogy a listadoboz nem rendelkezik *OnVScroll (OnVerticalScroll)* eseményjellemzővel. Most hát itt az alkalom, hogy készítsünk egy *OnVScroll* eseményjellemzővel rendelkező listadoboz komponenst.

Megoldás (15_KOMPONENSEK\SCROLLLIST.PAS)

Az új komponens a *TListBox* osztály leszármazottja lesz. Vezessük is rögtön be az *FOnVScroll* eseményjellemzőt. Típusának megválasztásában segít a beépített *TScrollBar* komponens. Helyezzünk el egy *TScrollBar* komponenst az űrlapunkon, és elemezzük az *OnScroll* eseményjellemzőjének típusát.
Íme mit mond róla a súgó:

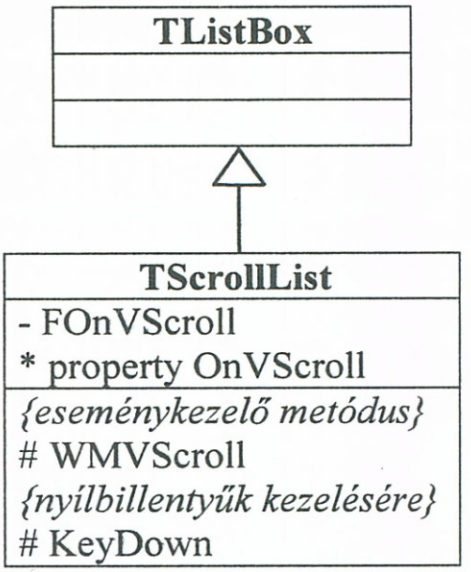
```
TScrollCode = (scLineUp, scLineDown, scPageUp, scPageDown, scPosition,
               scTrack, scTop, scBottom, scEndScroll);

TScrollEvent = procedure(Sender: TObject; ScrollCode: TScrollCode;
                         var ScrollPos: Integer) of object;

property OnScroll: TScrollEvent;
```

Tehát görgetéskor a következő információk fontosak:

- *Sender*: az esemény okozója
- *ScrollCode: scLineUp, scLineDown...* a görgetés kódja (egy sorral feljebb, lejjebb...)
- *ScrollPos*: hova kerül a görgetősáv a görgetés után

Ezeket az információkat kell nekünk is átadnunk az *OnVScroll* eseményjellemző felhasználójának (lekódolójának), így az új jellemző típusa *TScrollEvent* lesz.

Természetesen a hátterében itt is egy privát mező áll, az *FOnVScroll:TScrollEvent*.
A komponens *OnVScroll* eseményjellemzőre épített kódrészletnek minden egyes görgetéskor le kell futnia. Ennek érdekében írnunk kell egy eseménykezelő metódust, melyet a *WM_VSCROLL* Windows üzenetre irányítunk, és ebben fogjuk meghívni az *OnVScroll* jellemzőre épített kódrészletet. (A Windows rendszer üzeneteinek nevei megtalálhatók a WinSight segédprogram *Messages/Options* beállítóablakában, vagy a Delphi súgójában. Ezeket elég egyszer végignézni, később már „ráérzésre" kitaláljuk a neveiket.).
Általában a listadobozok billentyűzetről is görgethetők. Ha azt szeretnénk, hogy az *OnVScroll* eseményjellemző kódja ilyenkor is következzen be, akkor felül kell írnunk a *KeyDown* metódust: megvizsgáljuk benne a leütött billentyűt, a nyílbillentyűk esetén pedig meg kell hívnunk az *OnVScroll* kódját.



15.12. ábra. Osztálydiagram

Íme a kód:

```
unit ScrollList;

interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
     Dialogs, StdCtrls;

type
   TScrollList = class(TListBox)
      private
         FOnVScroll:TScrollEvent;
      protected
         procedure WMVScroll (var mes:TMessage); message WM_VSCROLL;
         procedure KeyDown(var Key:Word; shift:TShiftState);override;
      published
         property OnVScroll:TScrollEvent read FOnVScroll
                                          write FOnVScroll;
   end;

procedure Register;

implementation

{A WMVScroll metódus minden görgetéskor bekövetkezik. Ilyenkor először
végre kell hajtanunk az ősosztály görgetésre épített teendőit, majd
következhetnek saját „gondjaink": ha az OnVScroll jellemzőbe a fel-
```

használó kódot épített be, akkor az fusson le a megfelelő paraméterekkel. A paraméterek értékei a mes:TMessage üzenetrekordból származnak. A görgetés kódjának és a görgetősáv pozíciójának előállításához használjuk a Delphiben definiált TWMVScroll típust. (Sajnos a Delphi 3-as súgója kicsit „lebutult" az előző verziókéhoz képest. A Delphi 2-es súgójában minden nehézség nélkül megtalálható a típus leírása.) A pos változóra azért van szükség, mert az FOnScroll harmadik paramétere változó paraméter, tehát aktuális paraméterként csak változónevet lehet átadni.

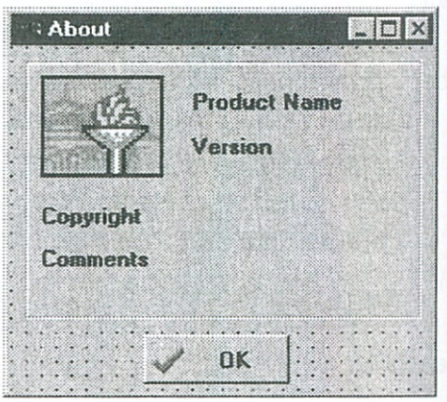
```
procedure TScrollList.WMVScroll(var mes:TMessage);
var  ScrollMes:TWMVScroll;
     pos:Integer;
begin
  Inherited;
  ScrollMes:= TWMVScroll(mes);
  pos:= ScrollMes.Pos;
  If Assigned(FOnVScroll) And
     (TScrollCode(ScrollMes.ScrollCode)<> scEndScroll) Then
     FOnVScroll(self,TScrollCode(ScrollMes.ScrollCode), pos);
end;

{Minden görgetés legalább két üzenetből áll (ez a Winsight-ban
vehető észre). Ha például kattintunk egyet a görgetés felfele nyilára,
akkor előbb keletkezik egy scLineUp kódú üzenet, majd ezt követi egy
scEndScroll kódú üzenet, mely a görgetés befejezését jelzi. Ha a befe-
jező üzenetre is meghívnánk az FOnVScroll kódját, akkor ez a kód
görgetésenként kétszer futna le.}
procedure TScrollList.KeyDown(var Key:Word; shift:TShiftState);
var pos:Integer;
begin
  Inherited KeyDown(Key, shift);
  pos:=1;
  If Assigned(FOnVScroll) Then
    begin
      {Ha a ↑ vagy ← nyilat ütötték le}
      If (Key In [VK_Up, VK_Left]) And
         {és ha a kijelöléssel pont a legfelső sorban tartunk}
         (ItemIndex=TopIndex) And
         {és nem a legelső elemen állunk...}
         (ItemIndex>0) Then
         FOnVScroll(self,scLineUp,pos)
      Else If (Key In [VK_Down, VK_Right]) And
         (ItemIndex-TopIndex+1 >= ClientHeight div ItemHeight ) And
         (ItemIndex<Items.Count-1)Then
         FOnVScroll(self, scLineDown,pos);
    end;
end;
```

```
procedure Register;
begin
   RegisterComponents('SajatKomponensek', [TScrollList]);
end;

end.
```

15.8 TAboutBox

Következzen most egy kicsit szokatlanabb feladat: készítsünk egy névjegyablak komponenst, melynek működése egyezzen meg a rendszer párbeszédablakokéval (*TOpenDialog*, *TSaveDialog*...). Úgy tervezzük meg, hogy egy bármilyen későbbi alkalmazásban a névjegyablak megjelenítése a következő lépésekből álljon:

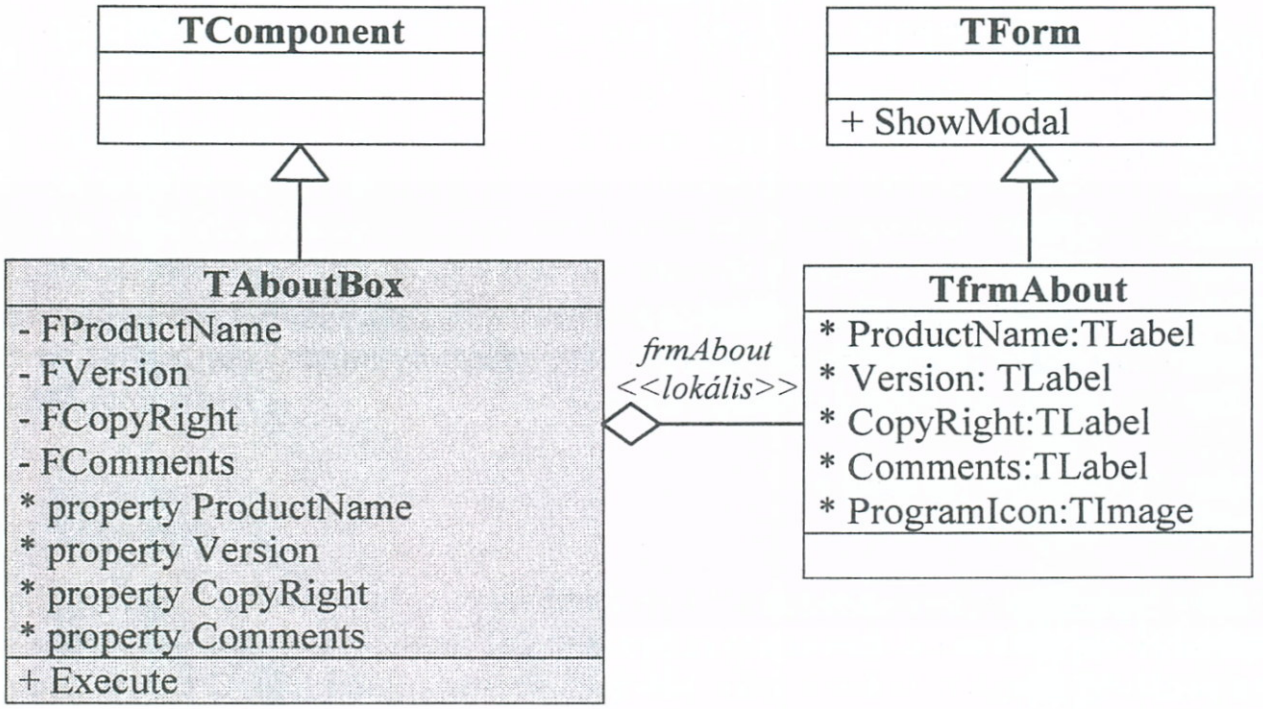
- Elhelyezünk az űrlapon egy *TAboutBox* komponenst.
- Beállítjuk jellemzőit *(ProductName, Version...)* az objektum-felügyelőben.
- Megjelenítjük az *Execute* metódusának segítségével.



15.13. ábra. A megjelenített névjegyablak

Megoldás (🖫15_KOMPONENSEK\ABOUTBOX.PAS)

A feladatspecifikációból egyértelműen kiderül, hogy komponensünkben szükség lesz a következő karakterláncos jellemzőkre: *ProductName*, *Version*, *Copyright* és *Comments*.
Új komponensünknél szükség van még egy *Execute* metódusra, ebben kell megjelenítenünk a névjegyablakot.
Látjuk már, hogy milyen mezői, jellemzői, metódusai lennének új komponensünknek, de még nem döntöttük el melyik legyen az ősosztálya. Habár megjelenítése után a komponens űrlapként viselkedik, mégsem származhat a *TForm* osztályból, hiszen akkor már tervezéskor is űrlap kinézete lenne. Most mi azt szeretnénk, ha tervezéskor egy kis négyzetben jelenne meg, mint a *TOpenDialog* is, és csak az *Execute* metódus hívásakor bontakozna ki űrlap formájában. Ez így bizonyára elég bonyolultnak tűnik, de valójában a megoldás rém egyszerű. Komponensünk származzon a *TComponent* osztályból, és tartalmazzon egy előzőleg megtervezett *TfrmAbout* űrlapot.



15.14. ábra. A *TAboutBox* komponens osztálydiagramja

Lépések:

- Tervezzük meg előbb az *frmAbout* űrlapot a *TAboutBox* űrlapminta alapján (*File/New...* menüpont *Forms* fül). Ne felejtsük el átnevezni az űrlapot *frmAbout*-ra.
- Mentsük le űrlapunkat az UABOUT.PAS állományba.
- Ezután hozzunk létre egy új komponenst: a neve legyen *TAboutBox*, őse pedig a *TComponent* osztály. Új osztályunkat a következőképpen kódoljuk le:

```
unit AboutBox;

interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
     Controls, Forms, Dialogs, uAbout; {csak így fogja ismerni a
     TfrmAbout típust}

type
  TAboutBox = class(TComponent)
     private
        FProductname, FVersion, FCopyRight, FComments: String;
     public
        procedure Execute;
     published
        property ProductName: String    read FProductName
                                        write FProductname;
        property Version: String read FVersion write FVersion;
```

```
        property CopyRight: String read FCopyRight write FCopyRight;
        property Comments: String read FComments write FComments;
    end;

procedure Register;

implementation

{A névjegyűrlap-objektum az Execute metódus lokális változója, csak a
metódus futásának időtartama alatt él. Ezért tüntettünk fel <<loká-
lis>> kapcsolatot az osztálydiagramon.}

procedure TAboutBox.Execute;
var frmAbout: TfrmAbout;
begin
   {A névjegyűrlap létrehozása}
   frmAbout:= TfrmAbout.Create(Self);
   Try
      With frmAbout Do
         begin
            {Adatok beállítása}
            Productname.Caption:= fProductName;
            Version.Caption:= fVersion;
            CopyRight.caption:= fCopyRight;
            Comments.caption:= fComments;
            {Megjelenítés}
            ShowModal;
         end;
   Finally
      {Végül felszabadítás}
      frmAbout.Free;
   End;
end;

procedure Register;
begin
   RegisterComponents('SajatKomponensek', [TAboutbox]);
end;

end.
```

Regisztrálja, majd tesztelje le ezt a komponenst is.

15.9 Súgó készítése egy saját komponenshez

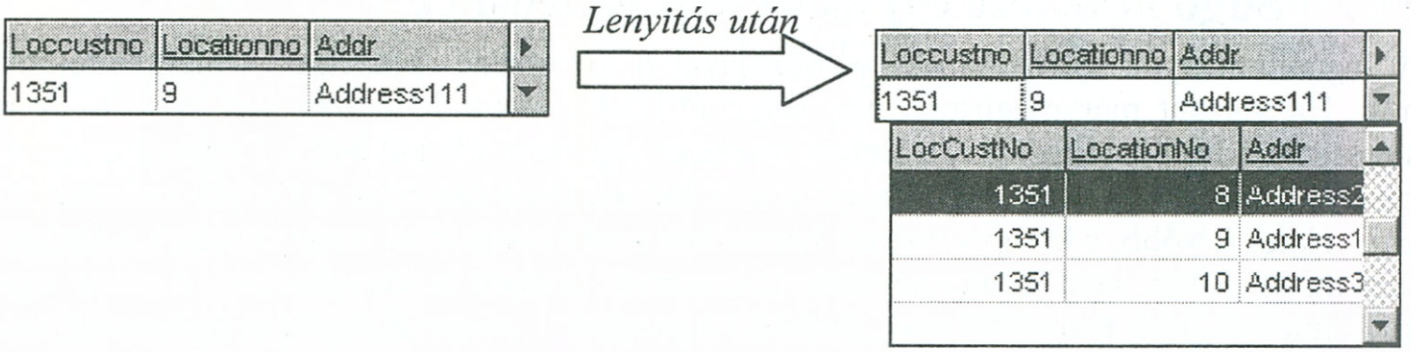
A súgóállományok készítésének technikájával a következő fejezetben ismerkedhetünk meg. Ha viszont már megírtunk egy súgót (*.HLP), akkor lehetőség van arra, hogy a kulcsszavait beépítsük például a Delphi rendszer súgójába. Ha például a *TAboutBox* komponens számára írunk egy súgót, majd ennek kulcsszavait beépítjük a Delphi kulcsszavai közé, akkor később, a komponensünk használata során nyugodtan megnyithatjuk a Delphi súgóját, és abban saját komponensünk kulcsszavai alapján is kereshetünk. Amikor egy saját kulcsszavunk témakörét meg szeretnénk jeleníteni, akkor a rendszer automatikusan megnyitja saját súgóállományunkat, így máris kézhez kaptuk a segítséget. Bővebben lásd a következő fejezetben.

15.10 Végszó

Az eddigiekben csak olyan komponenseket készítettünk, melyeknél egy már létező osztályt öröklési úton használtunk fel. Vannak olyan esetek is, amikor az új komponens csak a már létező Delphi komponensek átírásával készíthető el. Vegyük példának a beépített *TDBLookupComboBox* komponenst. Fix része egy szerkesztődobozból áll, melyben megjelenik az adatforrásként beállított adathalmaz egy mezőjének értéke. A *TDBLookupComboBox* osztály a *TEdit* osztály leszármazottja (*TDBLookupComboBox = Class(TEdit)...*).

Ha azt szeretnénk, hogy a kombinált lista fix részében több mezőt is meg tudjunk jeleníteni (táblázatos formában), és minden mezőnél az első sorban a neve, másodikban pedig az értéke (a tényleges adat) legyen látható (15.15. ábra), akkor egyértelmű, hogy a *TEdit* már nem megfelelő ősosztályként. Ezért meg kell változtatnunk az új osztály ősét, ami csak úgy lehetséges, hogy másolatot készítünk a *TDBLookupComboBox* forrásáról, és abban megfelelően módosítjuk az ősosztályt *(TSajatDBLookupCombo = Class (TÚjŐs))*. Az új ős lehetne a *TStringGrid*, vagy a *TDBGrid*, csak az a fontos, hogy táblázatos megjelenése legyen. **Az ős átírása sok súlyos következménnyel járhat** (például a forrásban mindvégig olyan mezőre hivatkozunk, mely a régi, *TEdit* ősben még megvolt, de az újban már nincs benne). Ezeket mind meg kell találni, és ki kell javítani. És ezzel még majdnem semmit sem csináltunk, hiszen a java csak most jön: a kombinált lista felső részében levő mezők közül húzzuk alá az indexelteket, ezen mezők értékeire lehessen inkrementálisan keresni, a lebomló listát egy beállítható szempont szerint lehessen szűrni stb.

Természetesen a továbbiakat itt nem lenne időnk és helyünk sem részletezni. Egy végkövetkeztetést azonban levonhatunk: *a komponensek fejlesztése igazi kihívást jelenthet a programozó számára.*



15.15. ábra. Listadoboz, melynek fix része egy kétsoros táblázat
(és még sok egyebet is tud)

Feladatok

TCaseEdit

Készítsen egy olyan szerkesztődoboz-komponenst, melynek legyen egy *Case* jellemzője. Lehetséges értékei: *caUpper*, *caLower*, *caNormal*. Ha tervezési időben *caUpper* értékre állítjuk, akkor minden begépelt betűt alakítson nagybetűssé. Ha *caLower*-re állítjuk, akkor mindent alakítson kisbetűssé, ha viszont *caNormal*-ban hagyjuk (ez legyen az alapértelmezett értéke), akkor minden bevitt karaktert hagyjon változatlanul. (Ez a feladat főképp a Delphi 1-essel rendelkező Olvasók számára érdekes, hiszen a későbbi verziókban ez a funkció már a *TEdit* osztályba lett beépítve. Még így sem árt azonban végiggondolni a megoldást.)

TEnterEdit (15_KOMPONENSEK\ENTREDIT.PAS)

Írjon egy olyan szerkesztődoboz komponenst, mely az Enter billentyű hatására átadja a fókuszt az űrlap (szülőkomponens) soron következő vezérlőelemének.

TCalculator

Készítsen egy számológép-komponenst a *TAboutBox* mintájára.

TClock

Fejlesszen ki egy *TClock* komponenst. Ez állandóan az űrlap jobb felső sarkában „csücsüljön", és – amint a neve is mondja – jelezze a pontos időt. A „lelkesebb" Olvasók kifejleszthetik a digitális és az analóg változatát egyaránt úgy, hogy egy jellemzővel lehessen válogatni a két megjelenítési mód között.

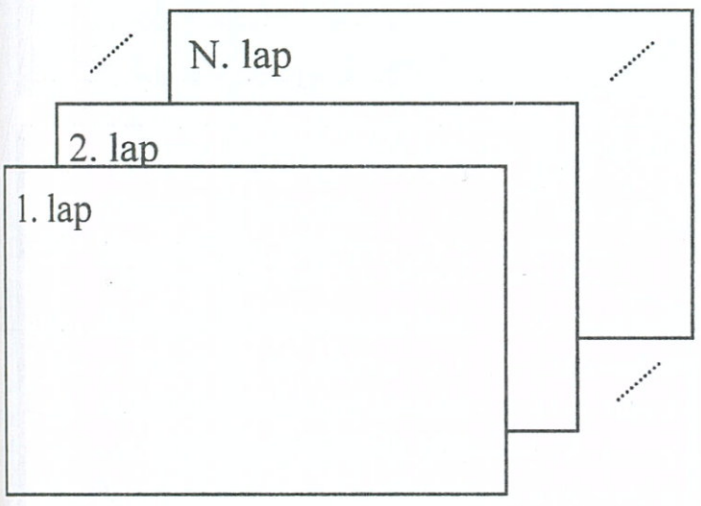
16. A súgó készítése

Egy windowsos alkalmazás szerves velejárója a súgóállomány is. Alkalmazásainkat úgy kell megírnunk, hogy használatukkor minél kevesebbet kelljen súgni, ha viszont valaki mégis segítségre szorul, akkor az legyen kéznél. Ebben a fejezetben bemutatjuk, hogyan lehet Windows súgót írni, és hogyan lehet ezt egy Delphi alkalmazásból megjeleníteni, használni.

16.1 A súgó szerkezete és használata

Mielőtt megismerkednénk a súgó készítésének „fortélyaival", tekintsük át együtt a súgó szerkezetét, valamint működésének, használatának általános szabályait.

Minden súgót a Windows rendszerbe beépített WINHLP32.EXE (16 bites Windowsban a WINHELP.EXE) programmal működtethetünk. A Word, az Excel, és a Delphi alkalmazások is tulajdonképpen csak egy (vagy több) .HLP kiterjesztésű állományt biztosítanak, ennek tartalmát a WINHLP32.EXE futtatásával jelenítik meg. A súgó tehát a HLP kiterjesztésű állomány(-ok)ból áll.

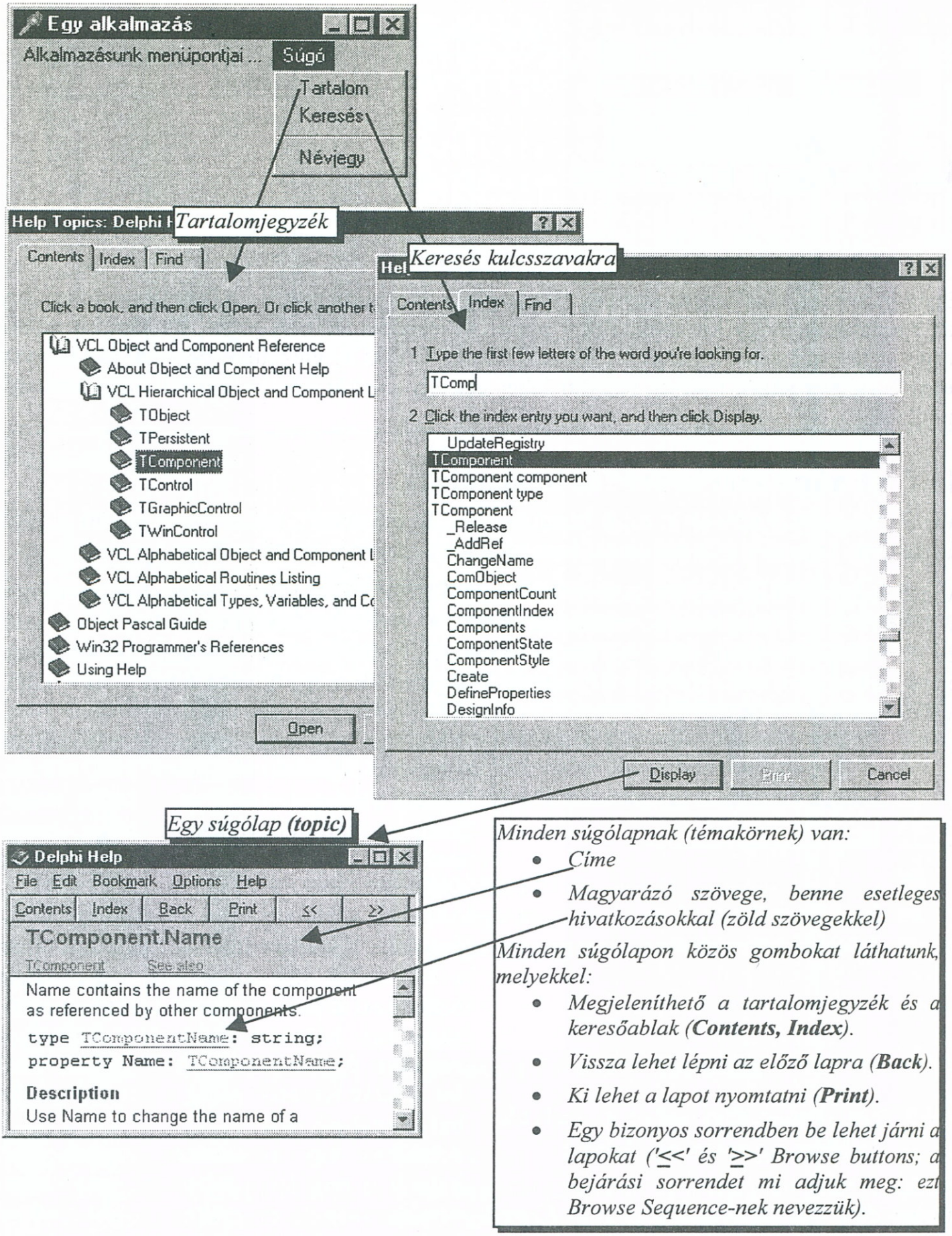


16.1. ábra. A súgó több lapból áll, melyből általában egyszerre egy látható

Minden súgóállomány egymásra hivatkozó lapokból (témakörökből, ***topic***) áll. Minden lapnak van egy címe (ezt a tetején láthatjuk), és van magyarázó szövege, benne esetleges hivatkozásokkal további lapokra (16.2. ábra).

A súgónak van egy összefoglaló tartalomjegyzéke is, mely hivatkozásokat tartalmaz a különböző súgólapok felé. Ugyanakkor biztosított a kulcsszavakra keresés lehetősége: egy kulcsszó begépelése közben a rendszer folyamatosan témaköröket kínál fel, a talált témakör pedig a *Display* gombbal jeleníthető meg.

Általában egy alkalmazás súgóját több helyről is meg lehet jeleníteni: meghívhatjuk menüből (általában a *Tartalom* és *Keresés* menüpontokkal), de gyakran találkozunk az ún. környezet-érzékeny (*Context Sensitive*) súgóval, azaz az F1 billentyű hatására az éppen kiválasztott elemről (lehet ez egy szöveg, egy űrlapelem stb.) kapunk információt. Mindezeket a 16.2. ábra foglalja össze.



16.2. ábra. Egy általános súgó szerkezete és használata

Az előbbi ábra a 32 bites környezetekben érvényes, ahol a súgót a WINHLP32.EXE működteti. Itt egyetlen ablak különböző oldalain lehet a tartalomjegyzéket megjeleníteni, valamint a kulcsszavas és az általános keresést lebonyolítani.
A 16 bites rendszerekben (WINHELP.EXE) csak sima – egy oldalas – keresőablakkal találkozhatunk, mely a kulcsszavakra való keresést valósítja meg. Itt is van tartalomjegyzék, de még nincs annyira kiemelt szerepe, mint a 32 bites változatokban. (A 32 bites súgóknál a tartalomjegyzék külön állományban helyezkedik el, a 16 biteseknél viszont a tartalomjegyzék a súgóállomány része, általában ez az első oldala.) Más, nem kulcsszavakra pedig nem is lehet rákeresni. A súgólap általános szerkezete megegyezik a későbbi verziókéval: itt is lehet nyomtatni, visszalépni..., itt is beállítható egy bizonyos sorrend (*Browse Sequence*), amiben a lapokat be tudjuk járni a '<<' és '>>' gombok segítségével.

16.2 A súgó készítésének lépései

A lényeg a HLP állomány elkészítésén van. Léteznek a szoftverpiacon speciálisan súgó készítésére kidolgozott programok. Ilyen például a ForeHelp, RoboHelp, Help Writer Assistant stb. Ha valaki sokszor ír súgót, akkor biztosan megéri valamelyiket megvásárolnia. Van azonban egy másik, kicsit munkásabb súgókészítő módszer is. Ez egy általánosan használható megoldás, nincs hozzá másra szükség, mint egy szövegszerkesztőre és egy súgófordítóra. (Ráadásul ezzel a módszerrel jobban megismerhetjük a súgó működését és szerkezetét.) A könyvben ezzel a módszerrel ismerkedhetünk meg. Lépések:

1. Először is meg kell írnunk, és le kell mentenünk a súgó szövegét egy vagy több RTF *(Rich Text Format)* kiterjesztésű állományba. Erre egy olyan szövegszerkesztőt kell használnunk, mely kezeli az **RTF formátumú** szövegeket és a **saját tervezésű** lábjegyzeteket (A lábjegyzetek jele általában egy szám, ezt a rendszer generálja, és az egymást követő lábjegyzeteknél automatikusan növeli: '1', '2'... Ezek az automatikusan számozott lábjegyzetek. Saját tervezésű egy lábjegyzet, ha annak jelét tetszőlegesen választjuk meg, például '*' vagy '#'. A súgóállományokban külön szerepük van a '#', 'K', '$' stb. lábjegyzeteknek, ezért a szövegszerkesztőnknek is ismernie kell ezeket.) Mindezen feltételeknek eleget tesz például a Word for Windows szövegszerkesztő. Példánkban mi ezt fogjuk használni.

2. A 32 bites környezetekben a súgónak többszintes tartalomjegyzéke is lehet, ezt láthatjuk a keresés ablakban a *Contents* lapon (16.2. ábra). Saját súgónk tartalomjegyzékét egy CNT kiterjesztésű szöveges állományban kell megadnunk. Elkészítésére használhatunk egy akármilyen szövegszerkesztőt (akár a Norton Editort is), vagy használhatjuk a 32 bites Delphi verziókba beépített Microsoft Help Workshopot (DELPHI\HELP\TOOLS\HCW.EXE).

   A 16 bites súgók tartalomjegyzékét az RTF állományban adjuk meg, így nincs szükség külön CNT állományra.

3. Harmadik lépésként a súgó projektállományát (HPJ) kell elkészítenünk. Ezzel már a súgó szövegállományok (*.RTF) lefordítását, azaz a HLP állomány létrehozását készítjük elő. A projektállomány is szöveges információkat tartalmaz, többek között azt is, hogy mely RTF állományokból származik a súgó szövege és honnan a tartalom-

jegyzéke, valamint azt is, hogy akarjuk-e súgónkban használni a '<<' és '>>' gombokat (*browsebuttons*), vagy nem. Ennek elkészítésére is alkalmas a Microsoft Help Workshop.

4. A következő lépés a súgó lefordítása. A 32 bites verziókban ez is a Help Workshop segítségével történik, míg a 16 bites Delphiben erre a célra a HCP.EXE vagy HC31.EXE programokat használhatjuk DOS parancssorból. A fordítás eredményeképpen elkészül a HLP kiterjesztésű súgóállomány. Működését leellenőrizhetjük már a Windowsban is: ha duplán kattintunk az állomány nevére, betöltődik a WINHLP32.EXE (WINHELP.EXE), és megjelenik súgónk szövege. Delphi alkalmazásunkba is a HLP állományt fogjuk beépíteni, ennek tartalomjegyzékét fogjuk megjeleníteni, és ennek lapjaira fogunk hivatkozni (lásd 16.4. pont).

Vegyük át ezeket a lépéseket egy konkrét példa kapcsán.

16.3 Feladat: a könyvnyilvántartó súgójának elkészítése

Írjunk egy súgót az előző fejezetekben létrehozott könyvnyilvántartónk számára. A súgó elkészítése után ezt be fogjuk építeni az alkalmazásunkba.

Megoldás (🖫 16_KONYVSUGO\)

16.3.1 A súgó szövegállományának (*.RTF) elkészítése

Az RTF állomány(-ok)ban minden egyes témakör (*topic*) fizikailag is külön oldalon jelenik meg. Kézzel fogunk közéjük oldaltöréseket (*Page Break*) elhelyezni. Minden egyes súgólap viszonylatában be kell állítanunk a következőket:

- **Azonosítóját (*Context String*):**

  Ez egy szöveg, mely a súgólap egyértelmű azonosítását szolgálja. Az azonosítót használjuk a tartalomjegyzékben és minden egyéb lapon is, ahol hivatkozni szeretnénk az adott témakörre. (Elvileg szám is lehetne (***Context ID***), de akkor a 10. lap után már valószínűleg belegabajodnánk az egészbe.)

- **Címét (*Title*):**

  Az itt megadott szöveg fog megjelenni az előzményekben (*History*) és a keresésnél is ezt fogja a rendszer találatként felkínálni. A súgólap címének nem kell kötelezően megegyeznie az ablak első sorában feltüntetett szöveggel.

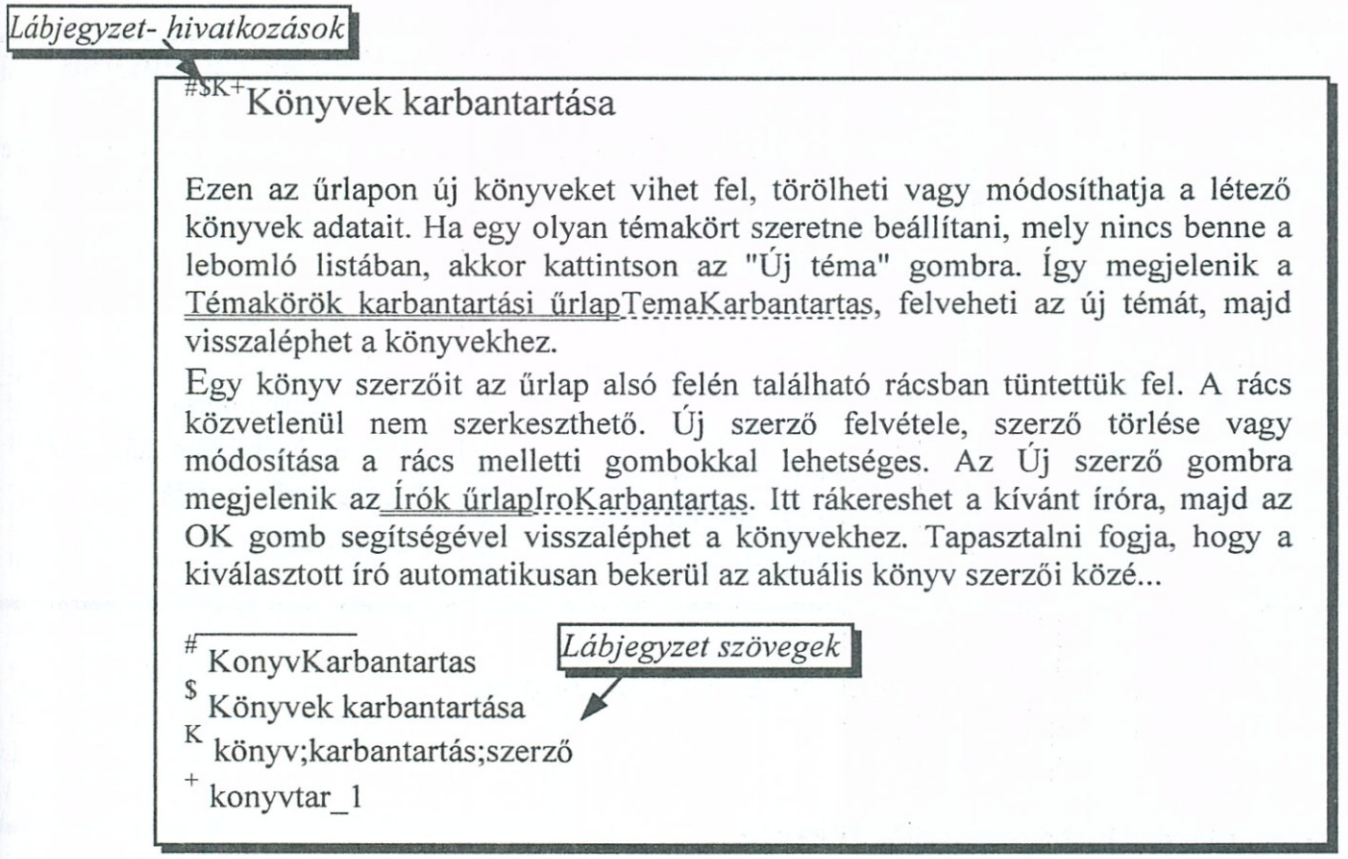
- **Kulcsszavait (*Keywords*):**

  Minden egyes lapnál ';'-vel elválasztva felsoroljuk azokat a fontosabb kifejezéseket (ún. kulcsszavakat), melyek a felhasználót keresés esetén erre az oldalra fogják vezetni.

- **Lapozási sorrend-azonosítóját (*Browse Sequence Number*):**

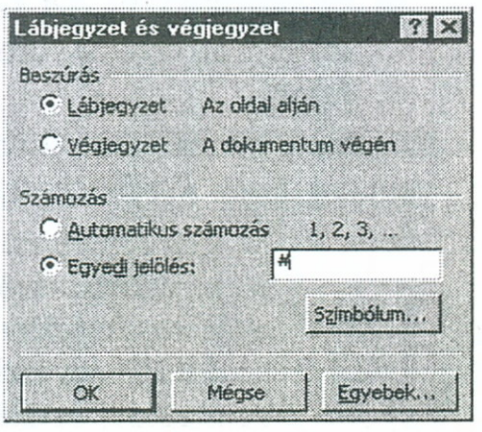
  Ha súgónkba beépítjük a '<<' és '>>' gombokat is, akkor minden egyes lapnak meg kell adnunk a bejárási sorszámát.

Egy súgótémakörrel kapcsolatban leggyakrabban ezt a négy információt kell beállítanunk. Az RTF állományban mindezeket speciális lábjegyzetek segítségével adhatjuk meg. Az azonosítót a '#', a címet a '$', a kulcsszavakat a 'K', a lapozási sorrendet pedig a '+' lábjegyzetekkel fogjuk specifikálni, amint ezt a 16.3. ábra is mutatja.



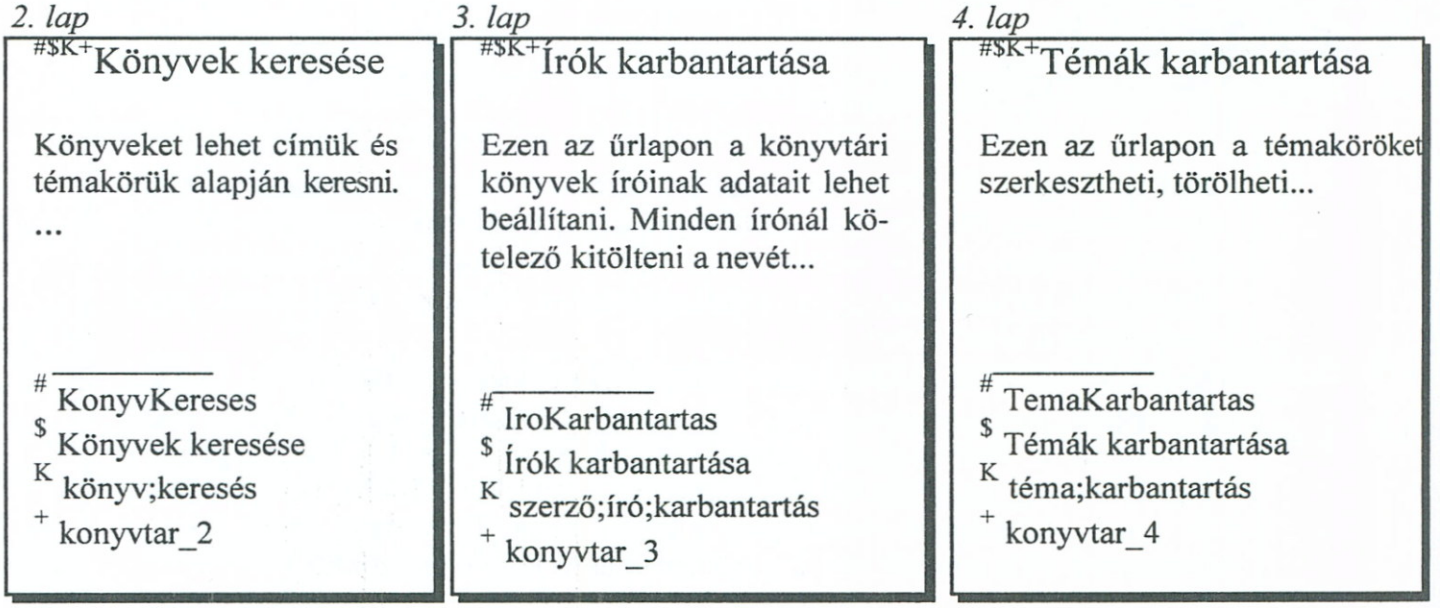
16.3. ábra. Egy súgólap felépítése
(Az aláhúzás egy másik lapra való hivatkozáshoz szükséges, lásd néhány oldallal később.)

Indítsuk el a Word szövegszerkesztőt, majd kezdjünk egy új dokumentumot. Nevezzük ezt el rögtön KONYVTAR.RTF-nek. (A mentésnél válasszuk ki a fájltípusok közül a Rich Text formátumot.)
Készítsük el súgónk első lapját a 16.3. ábra alapján. Figyelem, a lábjegyzeteket a *Beszúrás/Lábjegyzet...* menüponttal kell elhelyeznünk úgy, hogy a lábjegyzet jelét mi gépeljük be a megjelenő párbeszédablak *Egyedi jelölés* dobozába (16.4. ábra). Mivel négy lábjegyzetet kell elhelyeznünk, ezt a menüpontot négyszer fogjuk meghívni, más és más lábjegyzetjellel.



16.4. ábra. A lábjegyzet beszúrási párbeszédablak a Wordben

Egyelőre más oldalakra még nem hivatkozhatunk, hiszen még létre sem hoztuk ezeket. Írjunk még néhány oldalt a súgónkba (16.5. ábra).
Tipp: Nem kell minden oldal tetején a lábjegyzeteket újból és újból egyenként beszúrni. Elég, ha kijelöljük az első oldal lábjegyzeteit (a lábjegyzet-hivatkozásokat a lap címében), kimásoljuk ezeket a vágólapra (*Ctrl+C*), majd innen beillesztjük ezeket a későbbi lapokra (*Ctrl+V*). Így a lábjegyzet hivatkozások és ezek szövegei is bekerülnek az új lapra; ezek után elég a lábjegyzet szövegeket átírni az új lapnak megfelelően.

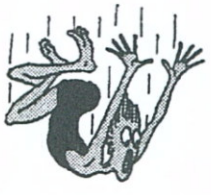


16.5. ábra. A többi súgólap

A hivatkozások (hot spots) elhelyezése

Ahhoz, hogy egy lapról megjeleníthessünk egy másikat a következők szükségesek:

- A hivatkozás szövegét húzzuk alá:
  - Duplán: ha azt szeretnénk, hogy az új lap egy teljes ablakban jelenjen meg
  - Pontozottan: ha azt szeretnénk, hogy az új lap egy kis „buboréksúgó" (előugró, magyarázó súgó) formájában jelenjen meg anélkül, hogy az aktuális lapot eltüntetné
- **Rögtön** a hivatkozás szövege után írjuk be a hivatkozott lap azonosítóját rejtett szövegként. Ezek a lefordított súgóban természetesen nem lesznek láthatók.

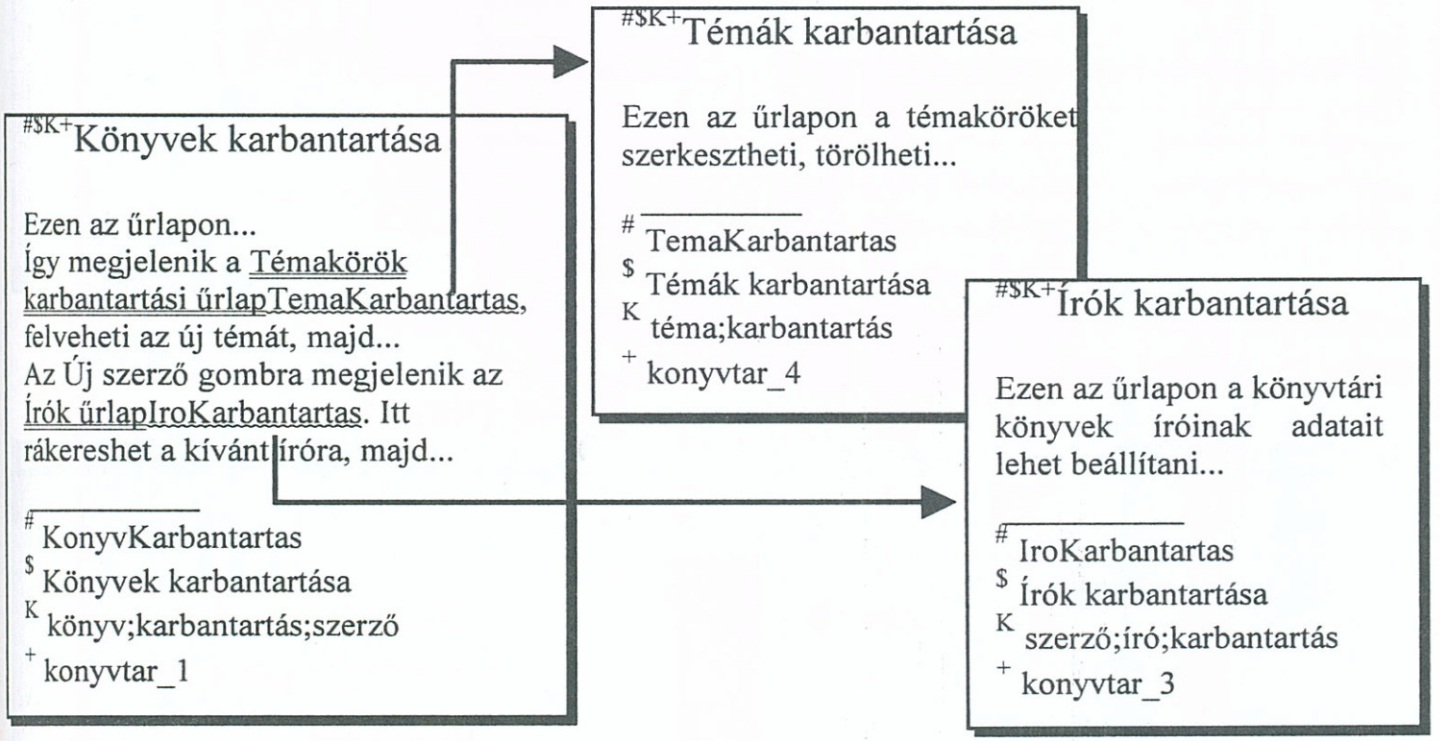


A hivatkozás-szövegek a súgó lefordítása után általában „bezöldülnek", és aláhúzottan jelennek meg. Ezt közvetlenül letilthatjuk úgy, hogy a rejtett azonosító elé egy szintén rejtett '%' vagy '*' jelet helyezünk el (a '%' jel hatására nem zöldül és alá sem lesz húzva, míg a '*'-ra nem zöldül be, de ugyanakkor aláhúzottan jelenik meg). Az ily módon „elrejtett" hivatkozásokat a felhasználó (súgónk olvasója) csak a *Ctrl+Tab* billentyűk lenyomva tartásával tudja megjeleníteni.

A formázásokat a *Formátum/Betűtípus...* menüpont segítségével végezzük el. Az aláhúzás típusát az *Aláhúzás* kombinált listából kell kiválasztanunk, ahhoz pedig, hogy egy szöveget elrejtsünk, elég a *Különlegességeknél* kipipálnunk a *Rejtett* jelölőnégyzetet.

Előfordulhat, hogy a rejtetten formázott szövegek már szerkesztéskor eltűnnek a képernyőről. Ahhoz, hogy ez ne történjen meg, válasszuk ki az *Eszközök/ Beállítások...* menüpontot. A megjelenő párbeszédablak *Megjelenítés* lapján pipáljuk ki a *Rejtett szövegrészek* jelölőnégyzetet.

Helyezzünk el súgónk első lapján két hivatkozást (16.6. ábra):



16.6. ábra. Hivatkozások elhelyezése. A rejtett szövegrészeket itt hullámos vonal jelzi.

Helyezzen el további hivatkozásokat más lapokon is, majd mentse le a munkáját RTF formátumban!

16.3.2 A súgó tartalomjegyzékének (*.CNT) elkészítése

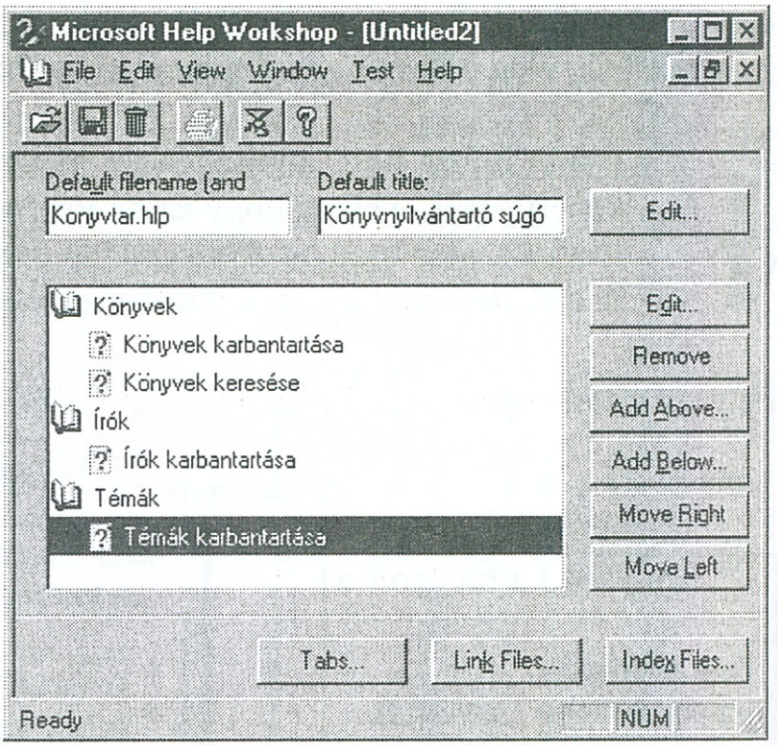
Ha 16 bites Windows alkalmazáshoz készítünk súgót, akkor a tartalomjegyzéket az RTF állomány első oldalára készítsük el. Ez csak hivatkozásokat fog tartalmazni a súgó többi lapjára.

A 32 bites környezethez készítendő súgó esetén a tartalomjegyzéket egy speciális, CNT kiterjesztésű állományban kell megadnunk. Ehhez indítsuk el a Microsoft Help Workshop segédprogramot, azaz a DELPHI\HELP\TOOLS\HCW.EXE-t, majd hajtsuk végre a következő lépéseket:

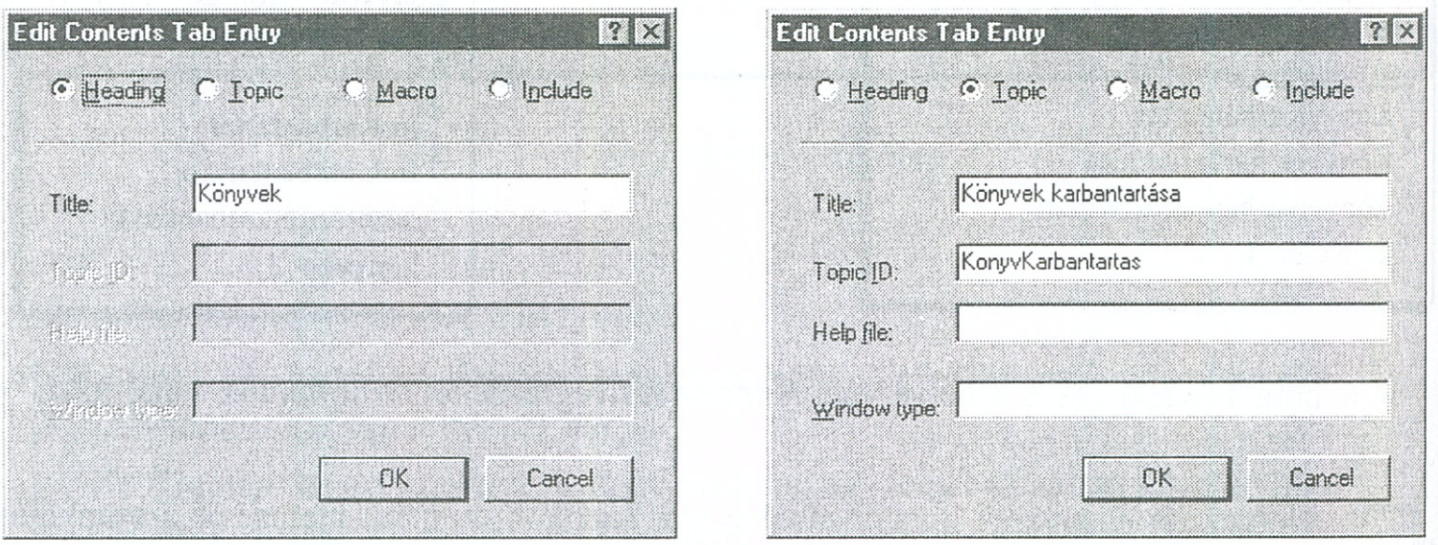
1. Hívjuk meg a *File/New...* menüpontot. Válasszuk a *Help Contents* opciót.

2. A megjelenő ablakban a *Default filename* és *Default Title* dobozokba gépeljük be az ábrán látottakat. A tartalomjegyzék bejegyzéseit az *Add Below...* gombra való kattintással hozhatjuk létre. Ha egy címet szeretnénk felvenni (a cím további témakörökre bomlik), akkor válasszuk a *Heading* stílust (16.8. ábra). A tényleges hivatkozásoknál fogadjuk el a *Topic* stílust. Ekkor be kell gépelnünk a bejegyzés szövegét (*Title*) és a hivatkozott lap azonosítóját is (*Topic ID*).

3. A szükséges bejegyzések létrehozása után mentsük ki munkánkat a KONYVTAR.CNT állományba a KONYVTAR.RTF mellé.



16.7. ábra. A tartalomjegyzék-szerkesztő



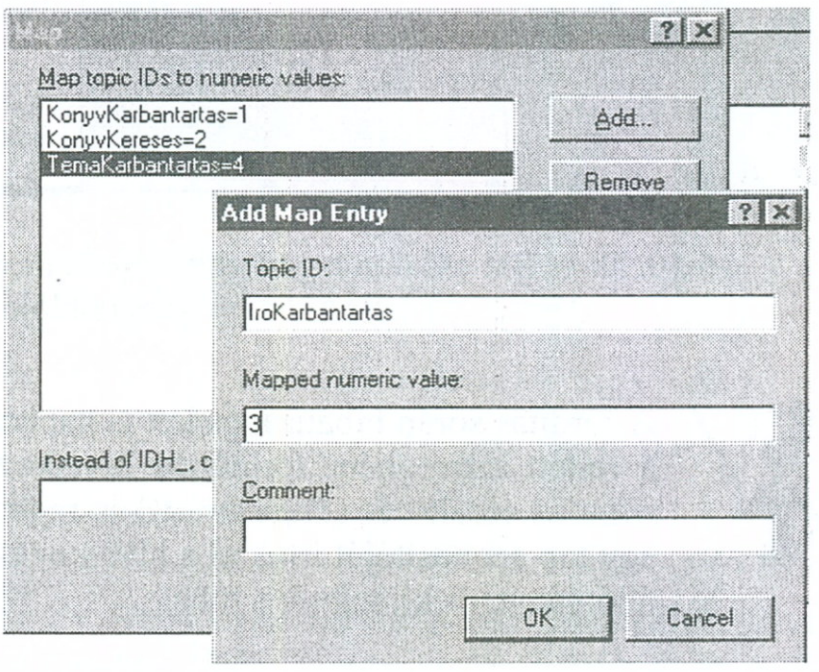
16.8. ábra. Bejegyzések felvétele a tartalomjegyzékbe

16.3.3 A súgó projektállományának (*.HPJ) elkészítése

Ezt is a Help Workshop segédprogrammal fogjuk létrehozni. Hívjuk meg a *File/New* menüpontot, és ezúttal válasszuk a *Help Project* opciót. Azonnal meg kell adnunk az új projektállomány nevét: KONYVTAR.HPJ. (Figyeljünk arra, hogy a *.RTF, *.CNT és *.HPJ állományok ugyanabban a könyvtárban legyenek.)

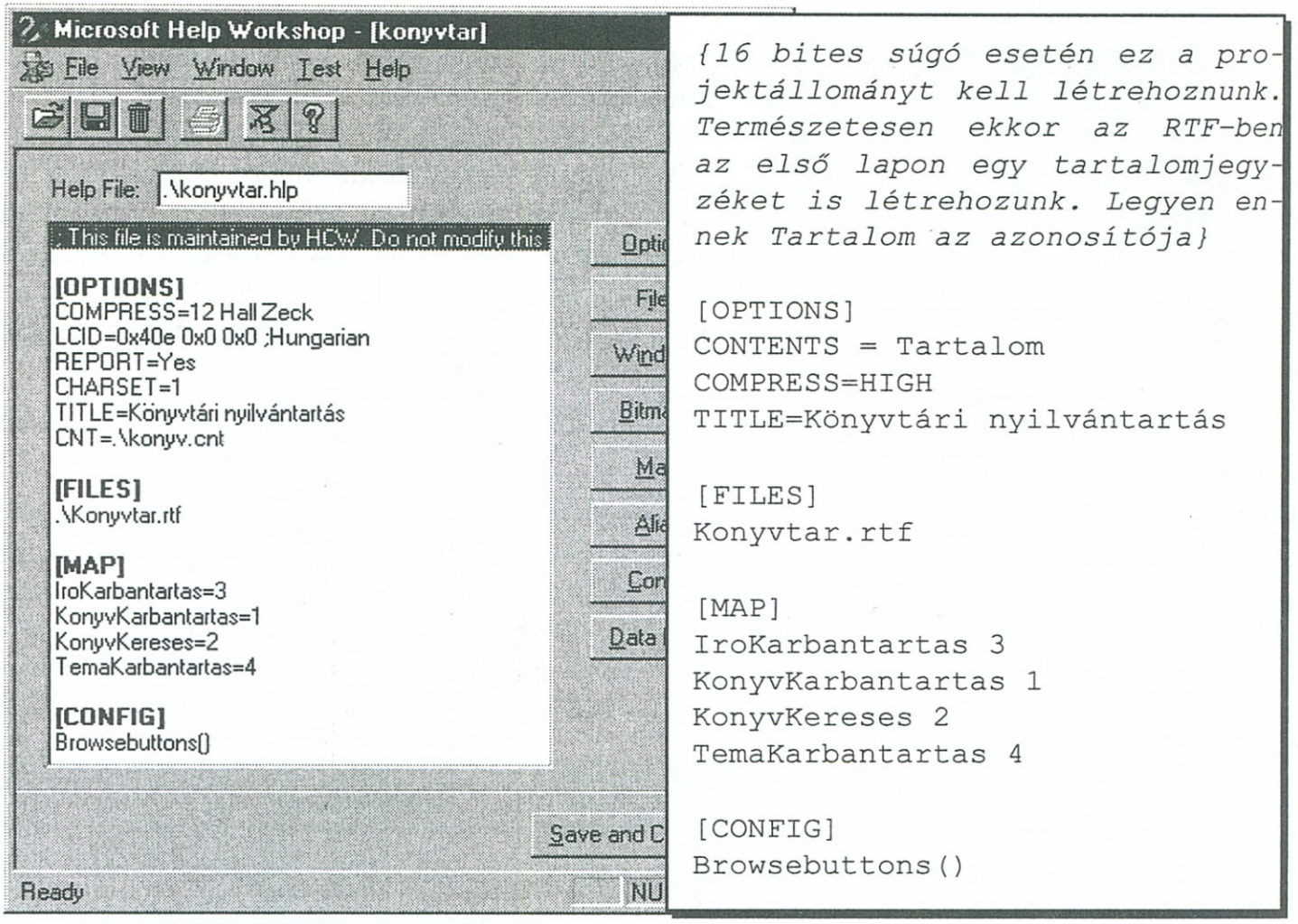
Kövessük a következő lépéseket:

- Kattintsunk az *Options* gombra. A megjelenő párbeszédablak *General* lapján a *Help Title* dobozba írjuk be a súgó címét. Ha a tartalomjegyzék az RTF állományban lenne (16 bites súgó), akkor a *Default Topic* dobozba a tartalomjegyzék lapjának azonosítóját kellene beírnunk. Mivel a 32 bites súgónkban a tartalomjegyzék külön állományban található, a *Default Topic* dobozba most semmit sem írunk. Váltsunk a *Compression* lapra, majd állítsuk maximumra a súgó fordításakor történő tömörítési arányt. A *Files* lapon válasszuk ki a KONYVTAR.RTF és KONYVTAR.CNT állományokat a *Rich Text Format (RTF) files* és a *Contents file* dobozokban. Végül zárjuk be az *Options* párbeszédablakot.
- Ha súgónk több RTF állományból áll össze, akkor a *Files* gombra kattintva állítsuk be a többi RTF állományt is.
- A következő lépésben a szöveges lapazonosítóknak egy-egy azonosítószámot fogunk megfeleltetni. Erre azért van szükség, mert igaz ugyan, hogy a lapazonosításra eddig kiválóan használhattuk a szövegeket, delphis alkalmazásunkban viszont ezekre már nem hivatkozhatunk. Minden Delphi komponens (gomb, szerkesztődoboz...) rendelkezik egy *HelpContext* nevű és *Integer* típusú jellemzővel. Ha ebbe beírjuk a megfelelő súgólap azonosítószámát, akkor az F1 billentyű lenyomásakor ez a lap fog megjelenni. Így tudunk tehát környezet-érzékeny súgót létrehozni delphis alkalmazásainkban. Ehhez viszont az is szükséges, hogy minden lapazonosítónak megfeleltessünk egy egyedi számot. Ezt a *Map* gomb segítségével tehetjük meg. A megjelenő párbeszédablakban (16.9. ábra) feleltessünk meg a négy lapazonosítónak négy számot.



16.9. ábra. A lapazonosítók „számosítása”

- A következő lépés a '<<' és '>>' gombok konfigurálása lesz. Ennek érdekében kattintsunk előbb a *Configuration*, majd utána az *Add* gombra. A megjelenő ablakba írjuk be: *BrowseButtons()*. Ezzel tulajdonképpen egy makrót hívunk meg, melynek eredményeképpen a súgólapok közötti lépegető gombok beépülnek súgóállományunkba.
- Ha ezzel is végeztünk, akkor fordítsuk le a súgóállományt a *Save and Compile* gomb segítségével. A 16 bites súgóhoz a 16.10. ábra jobb oldalán látható projektállományt hozzuk létre (például a Norton Editorban), majd a DOS parancssorból fordítsuk le: HCP KONYVTAR.HPJ.



16.10. ábra. Bal oldalon a generált projektállomány, jobb oldalon ennek 16 bites megfelelője

A fordítás során hibaüzeneteket is kaphatunk. Egy tipikus hiba például az, hogy a rejtett szövegként formázott hivatkozások miatt néhány sorvégi Enter, sőt néha az oldaltörés (*Page break*) is rejtett lesz. De ugyanígy előfordulhat, hogy egy lap azonosítóját írtuk el a hivatkozásban. Ilyenkor újból megnyitjuk az RTF állományt, és kijavítjuk a hibát.

Figyelem! Ha az RTF állományt **a Word 97**-es verziójában készítjük el, akkor fordításkor nagy valószínűséggel találkozni fogunk a '*Help file corrupted at...*' hibaüzenettel. Az üzenetben megjelenő szám az állomány utolsó bájtjára hivatkozik. A Word 97-es az RTF állományok végére **egy** kapcsos zárójel helyett **kettőt** helyez el. Ha tehát ebben készítjük el súgónkat, akkor egy másik szövegszerkesztő (például Norton Editor) segítségével lépjünk be az RTF állományba, és töröljük ki az állomány legvégén levő kapcsos zárójelet. Így súgószövegünk már igazi RTF formátumban lesz, a súgófordító is felismeri.
Aki irtózik az RTF állomány utólagos kézimunkálásától, a súgószöveg megszerkesztésére használja a Word 6-os vagy 95-ös verzióit.

A sikeres fordítás eredményeképpen elkészül a KONYVTAR.HLP állomány. Kipróbálhatjuk a Windowsból duplakattintással, vagy a Help Workshopból a *File/Run Winhelp...* menüpont segítségével.

16.4 A súgó használata Delphi alkalmazásainkban

Nyissuk meg az előző fejezetekben készített könyvnyilvántartót. Egészítsük ki a menüjét még egy utolsó menüponttal: *Súgó*. Két almenüpontja lesz: *Tartalom* és *Keresés*.
A Delphi alkalmazásban be kell állítanunk a használt súgóállományt. Ezt megtehetjük kódból (*Application.HelpFile:= 'KONYVTAR.HLP'*), de a *Project/Options* menüpont hatására megjelenő párbeszédablak *Application* oldalán is (*Help File :='KONYVTAR.HLP'*).
Ezek után beállíthatunk komponenseiknél környezet-érzékeny súgót. Állítsuk a menüpontok *HelpContext* jellemzőjét a számazonosítókra. Futáskor, egy adott menüpont feletti F1 billentyű leütése az ily módon beállított súgólapot fogja megjeleníteni.

Menüpont	HelpContext
Könyvek/Karbantartás	1
Könyvek/Keresés	2
Írók/Karbantartás	3
Témák/Karbantartás	4

A Tartalom és Keresés menüpontokra a következő utasításokkal jelenítjük meg a súgót:

```
procedure TfrmFo.Tartalom1Click(Sender: TObject);
begin
  Application.HelpCommand(Help_Finder,0);
  {16 bites súgóhoz: Application.HelpCommand(Help_Contents,0);}
end;

procedure TfrmFo.Kereses1Click(Sender: TObject);
begin
  Application.HelpCommand(Help_PartialKey,LongInt(StrNew('')));
end;
```

Mindkét metódusban a *TApplication* osztály *HelpCommand* metódusát használjuk. Első paraméterként egy parancsot vár (*Help_Contents*, *Help_Finder*, *Help_HelpOnHelp*...). A második paraméter értelmezése az első értékétől függ: például a kulcsszavas keresésnél a parancs: *Help_PartialKey*, a második paraméterben pedig át kell adnunk a keresett szövegrészt *PChar* karakterláncként.

Próbálja ki az alkalmazásban a környezet-érzékeny súgót, valamint a menüpontokat is.

16.5 Tippek, tanácsok

Íme zárszóként néhány tipp és tanács:

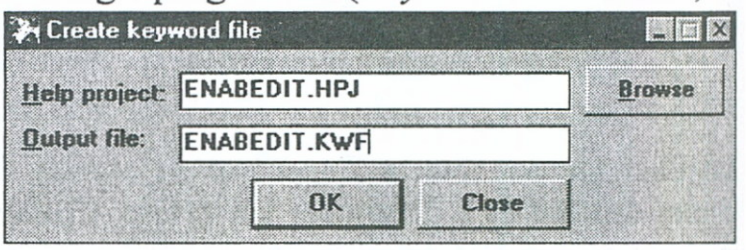
- Nagyobb rendszerekben, vagy egy-egy komolyabb komponensnél könnyen előfordul, hogy a súgó szövege is terjedelmessé válik. Kb. 50 oldal után már a szerkesztése is lelassul, és az áttekinthetősége is jelentősen csökken (a lelassulás függ a beágyazott ábrák mennyiségétől is). Ilyenkor **ajánlatos több RTF állományra szétosztani a súgószöveget**. A logikailag összetartozó RTF állományokat a súgó projektállomány (HPJ), valamint fordítás után a HLP kapcsolja össze. Természetesen az egyes RTF állományokban változatlanul hivatkozhatunk más RTF-beli súgólapokra, mivel a hivatkozásokat csak a fordító értelmezi, neki pedig a HPJ-ből tudomása van súgórendszerünk felépítéséről.
- **Lehetőség van a teljesen különálló HLP állományok egységes kezelésére is.** E technikának legelterjedtebb alkalmazása a saját (esetleg megrendelésre írt) komponensek súgóállományának beépítése a Delphi rendszer súgójába.

  A 16 bites Windowsban a HLP állományokat egy közös kulcsszólista kapcsolja össze (A Delphi 16 bites verziójának súgórendszere eleve több HLP állományból áll, de ugyanakkor a kulcsszavak együtt vannak a DELPHI.HDX állományban). Ha keresésnél egy másik állománybeli témakörre bukkanunk, akkor a rendszer azt azonnal berántja, témakörét megjeleníti.

  A 32 bites Windows verziókban a CNT állomány segítségével kapcsolhatunk össze több HLP fájlt. Itt a kulcsszavakra való keresésen kívül alkalmunk nyílik arra is, hogy a különböző HLP állományoknak egy közös tartalomjegyzéket készítsünk. Nézzük a részleteket! Tegyük fel, hogy van egy ENABEDIT.HLP súgóállományunk az előző fejezetben fejlesztett *TEnabEdit* komponensünkhöz. Nézzük, mit kell tennünk annak érdekében, hogy ez beolvadjon a Delphi rendszer súgójába:

  16 bites Windowsban: (16_KOMPSUGO\WIN16\ENABEDIT.*)

  - Indítsuk el a KWGEN.EXE segédprogramot (***Keyword Generator***, a DELPHI\HELP könyvtárban található). Ezzel kigyűjtjük a kulcsszavakat a saját HLP állományunkból az ENABEDIT.KWF állományba (16.11. ábra).



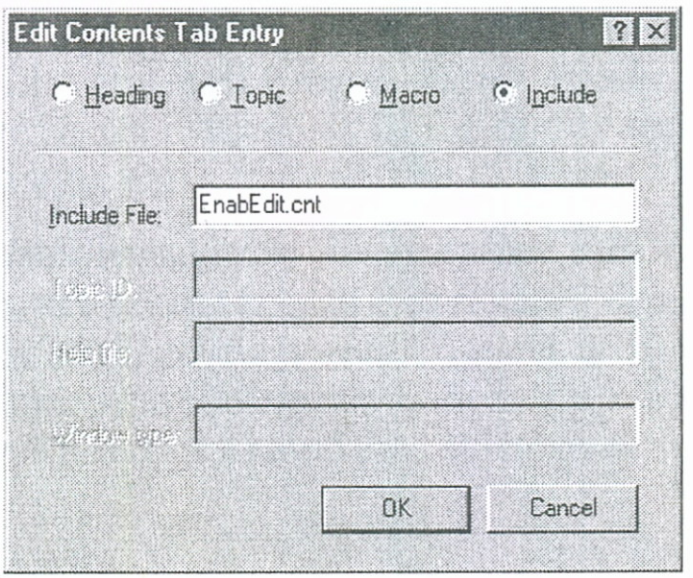
16.11. ábra. Kulcsszavak kigyűjtése Win3.x-ben

  - Másoljuk át az ENABEDIT.HLP és ENABEDIT.KWF állományokat a Delphi súgóállományai mellé (...\DELPHI\BIN).
  - Indítsuk el a HELPINST.EXE segédprogramot; ezzel az előző lépésben kigyűjtött kulcsszavakat beépítjük a Delphi kulcsszavai közé. Nyissuk meg a DELPHI.HDX állományt, majd az '+' gombbal adjuk hozzá a

ENABEDIT.KWF állományt. Mentés után nyissuk meg a Delphi súgóját (DELPHI.HLP). Tapasztalni fogjuk, hogy a kulcsszavak listája már a sajátjainkat is tartalmazza.

↳ **32 bites Windowsban: (💾 16_KOMPSUGO\WIN32\ENABEDIT.*)** Feltételezzük, hogy súgónkhoz ENABEDIT.CNT is tartozik. Lépések:

- Másoljuk át súgónk állományait (*.HLP, *.CNT) a Delphi rendszer súgóállományai mellé (...\Delphi3\HELP).
- Indítsuk el a HCW.EXE segédprogramot.
- Nyissuk meg a Delphi rendszer tartalomjegyzék állományát (...\DELPHI3\HELP\DELPHI3.CNT).
- Kattintsunk a jobb alsó sarokban található *Index Files...* gombra. Az előbukkanó párbeszédablak *Add* gombjával adjuk hozzá az ENABEDIT.HLP állományt. Ezzel a kulcsszavakat már össze is fésültük.
- Ha a saját tartalomjegyzékünket meg szeretnénk jeleníteni a Delphi tartalomjegyzékében, akkor az *Add Below...* vagy az *Add Above...* gombokkal vegyünk fel egy új tartalomjegyzék-bejegyzést. Ennek típusa *Include* legyen. Gépeljük be az ENABEDIT.CNT állomány nevét (16.12. ábra).
- Mentsük le a DELPHI3.CNT állományt. Máris kipróbálhatjuk az eredményt.



16.12. ábra. Tartalomjegyzékünk hozzáadása

- **Súgóinkba képeket is beilleszthetünk, sőt a különböző ábrákat hivatkozásokhoz is használhatjuk.** Egyszerűen húzzuk ezeket alá (duplán vagy pontozottan), ennek hatására fordítás után *hot spot*-ként fognak működni. Ha azonban súgónkban sok képre van szükség, akkor a képek beillesztésére használjuk a speciális, RTF formátum által felismert mezőutasításokat: a *brc*, *bml* és *bmr* (Például *{bmc KEP.BMP}*). Ha ezt nem tesszük, akkor könnyen előfordulhat, hogy a túlnövekedett RTF állományokat a 16 bites fordító nem képes lefordítani. További információkat a 16 bites verziókban a HELPREF.HLP állományban, a 32 bitesekben pedig a HCW súgójában találhatunk.

17. A Delphi alkalmazások telepítése

Ebben a fejezetben a Delphiben elkészített alkalmazásaink telepítését mutatjuk be, megismerkedünk a 32 bites Delphi környezetekhez tartozó InstallShield Express telepítő-készítő segédprogrammal.

17.1 Általános tudnivalók

A **nem adatbázisos alkalmazásoknál** a célgépre elég elvinnünk a program állományait: az EXE állományt, az esetlegesen használt saját DLL-eket, a HLP súgóállományt. Ilyenkor alkalmazásunk futásához nem szükséges egyetlen Delphi rendszerbeli DLL állomány sem (kivéve akkor, ha futás idejű csomagokat – *runtime packages* – használ, lásd 15. fejezet).

A Delphiben fejlesztett **adatbázisos alkalmazások** futás közben támaszkodnak a BDE bizonyos állományaira. Hogy pontosan melyekre, az a kezelt adatbázis formátumától függ. Ha tehát működtetni szeretnénk egy adatbázisos alkalmazást egy másik, Delphi nélküli gépen, akkor arra telepítenünk kell a saját alkalmazásunk állományain kívül még a BDE egy részét is. Egy általános adatbázisos alkalmazás futásához a következők szükségesek:

- Az alkalmazás EXE állománya
- Saját DLL-ek, futás-idejű csomagok (ha vannak)...
- Az alkalmazás adatállományai (ha ezek még nem léteznek a célgépen)

  Ha például egy elavult alkalmazás helyett most egy újat, korszerűbbet készítetünk, amely a régihez tartozó létező adatbázist használja, akkor nyilván telepítéskor nem kell az adatokkal foglalkoznunk, hiszen ezek már a célgépen jelen vannak. Ha viszont egy vadonatúj alkalmazást fejlesztünk, amiben az adatbázist is mi hozzuk létre, akkor ezt telepítéskor a célgépre is el kell vinnünk. Az adatbázis célgépen való létrehozása vagy az adatállományok átmásolásából, vagy az adatbázist generáló SQL szkriptállomány lefuttatásából áll. Az első megoldás a fájl-szerver, míg a második a kliens/szerver architektúrájú adatbázisoknál használatos.
- A súgórendszer állományai
- A BDE egy része

Ezeket nem elég egyszerűen felmásolni a célgépre, ott még létre kell hoznunk a regisztrációs adatbázisban *(registry)* a megfelelő bejegyzéseket is. Természetesen a célgépen a használt álneveket is konfigurálnunk kell. Minden jel arra mutat, hogy a telepítőprogram elkészítése fáradságos munkának ígérkezik: a BDE állományok kiválogatása, a telepítőlemezek elkészítése (tömörítés, lemezekre osztás...), a regisztrációs adatbázis bejegyzé-

seinek létrehozása... Sajnos a **16 bites Delphiben** írt alkalmazásokhoz saját maguknak kell ezeket a lépéseket végigszenvednünk. A **32 bites verzióknál** rendelkezésünkre áll az **InstallShield Express** nevű program, mely ezt a fáradságos munkát elvégzi helyettünk. (Ezt a segédprogramot külön kell feltelepíteni a Delphi CD-ről.)
Ebben a fejezetben az InstallShield Express program használatát mutatjuk be egy feladaton keresztül:

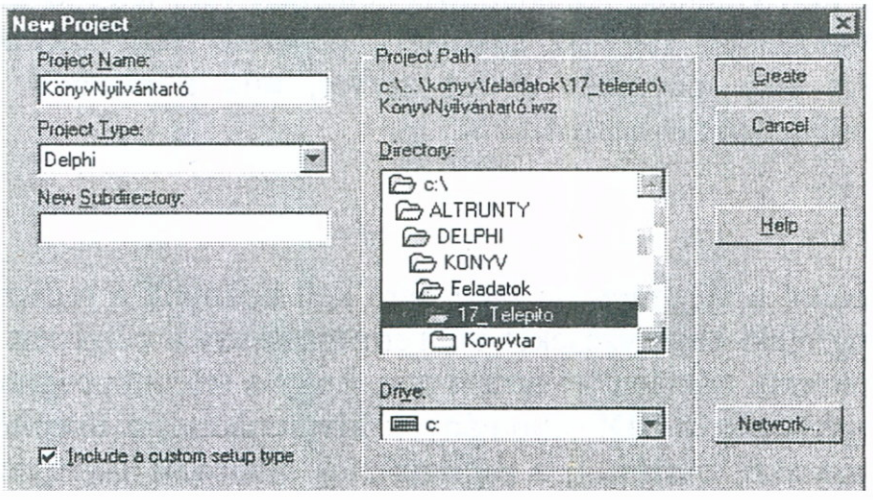
Készítsünk telepítőkészletet az előző fejezetekben fejlesztett könyvnyilvántartó alkalmazás számára.

Megoldás (🖫 17_TELEPITO\KONYVNYILVANTARTO.IWZ)

Először gondoljuk végig a teendőket!
Könyvnyilvántartó programunk EXE és súgójának állományai a 16_KONYVSUGO könyvtárban találhatók, a használt adatállományok pedig az ADATOK\KONYVTAR-ban (ne felejtsük el, hogy az alkalmazásban a *DBKonyvtar* álnévvel hivatkozunk az adatokra). Ezeken kívül szükség lesz a BDE bizonyos állományaira is a DELPHI\BDE könyvtárból. Feltételezzük, hogy az adatállományok még nem léteznek a célgépen, ezeket mi terveztük meg, mi hoztuk létre, és most át kell másolnunk ezeket a célgépre is.
Készítsük el alkalmazásunk telepítőkészletét. Lehessen általános (*typical*), minimális (*compact*) vagy akár egyéni (*custom*) telepítést kérni, és természetesen programunkat lehessen később eltávolítani az *UnInstall*-lal. Mindezeket az opciókat helyesen be kell majd állítanunk az *IstallShield* programban.

17.2 Az InstallShield Express indítása

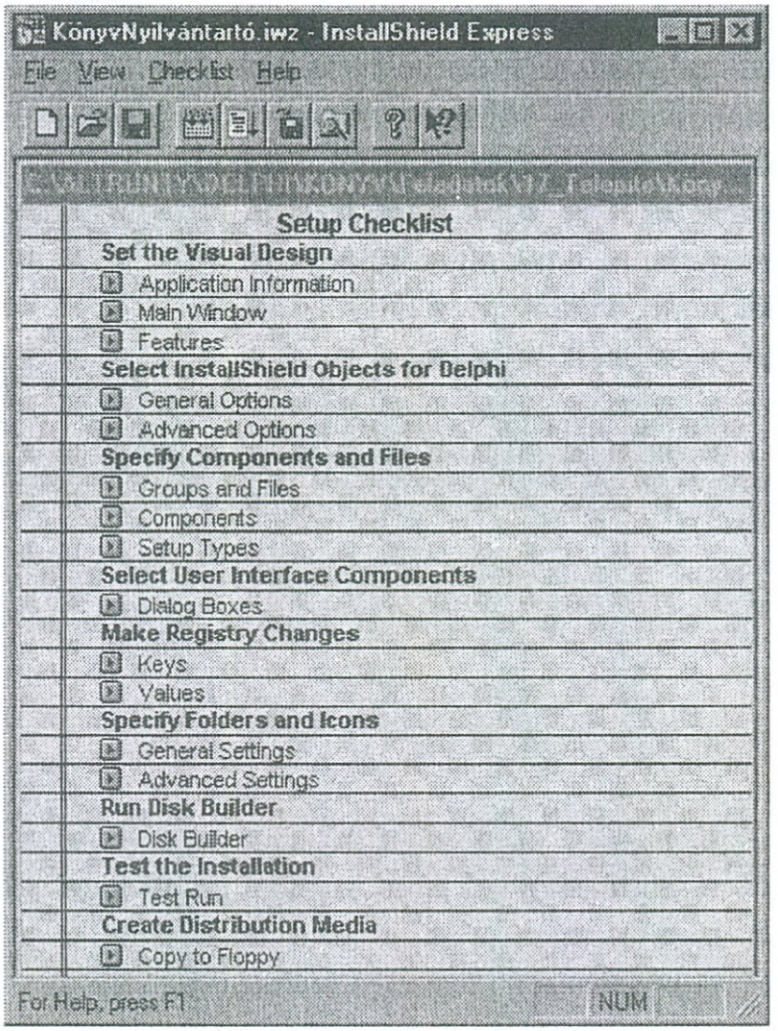
Indítsuk el az InstallShield Express programot. Válasszuk a *Create a new Setup Project* opciót, majd adjuk meg az elkészítendő projektállomány nevét és útvonalát (17.1. ábra). A program létrehoz egy KÖNYVNYILVÁNTARTÓ.IWZ állományt (telepítő projektállományt) a beállított útvonalon, ebben tárolja a telepítőkészlet beállításait.
Figyelem! Ha egyéni telepítési lehetőséget is be szeretnénk építeni a telepítőkészletünkbe, akkor már most a legelején ki kell pipálnunk az *Include a custom setup type* jelölőnégyzetet.



17.1. ábra. A telepítő projektállomány adatai

A frissen létrehozott projektállomány ablakában a további teendőink, beállítanivalóink láthatók (17.2. ábra):

- *Set the Visual Design:* a telepítőprogram megjelenését befolyásoló adatok;
- *Select InstallShield Objects for Delphi:* itt kell beállítanunk a BDE használandó állományait;
- *Specify Components and Files:* itt meg kell adnunk az alkalmazás futásához szükséges összes állományt. Ezeket különböző csoportokba sorolhatjuk, valamint beállíthatjuk, hogy melyek szükségesek a minimális, általános és egyéni telepítéshez;
- *Select User Interface Components:* milyen párbeszédablakok jelenjenek meg a telepítés időtartama alatt;
- *Make Registry Changes:* a telepítendő alkalmazás regisztrációs adatbázis beállításai;
- *Specify Folders and Icons:* a telepítendő alkalmazás programcsoportjának és ikonjának beállításai;
- *Run Disk Builder:* az eddigi beállítások alapján tömöríti az állományokat és elkészíti a telepítőlemezek tartalmát;
- *Test the Installation:* próbatelepítés;
- *Create Distribution Media:* az elkészített telepítőkészlet állományait itt másolhatjuk fel lemezekre.

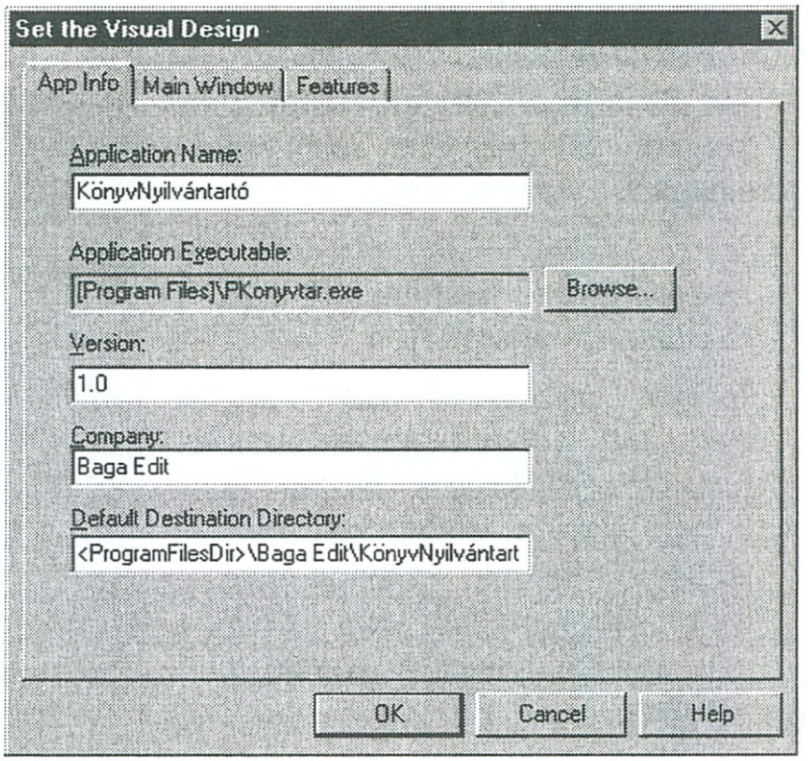


17.2. ábra. A telepítőkészlet elkészítésének lépései

17.3 A telepítő külalaki adatai

Kattintsunk az ***Application Information*** sorra. A megjelenő párbeszédablak (17.3. ábra) első oldalán meg kell adnunk az alkalmazás nevét, az EXE állományt, a verziószámot és a fejlesztő cég nevét. Az alapértelmezett telepítési könyvtár automatikusan ezekből épül föl.

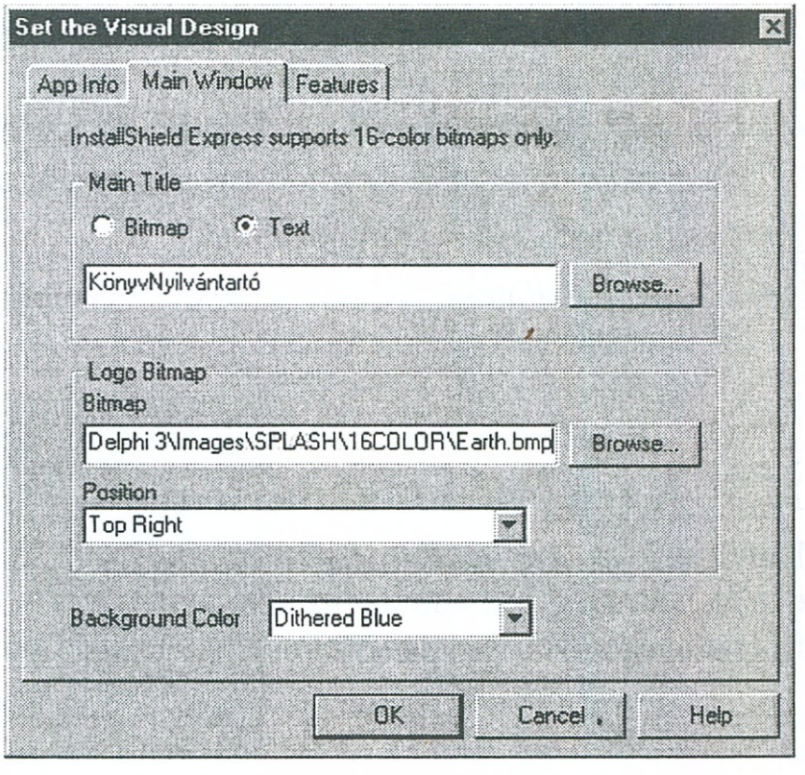
Az alkalmazás neve (*Appliction Name*) akár több szóból is állhat, de maximum 80 karakteres lehet. Természetesen használhatunk benne ékezeteket is. Ez a karakterlánc fog megjelenni a telepítés különböző ablakaiban, valamint ez adja az alapértelmezett célkönyvtárat is.



17.3. ábra. A telepítendő alkalmazás adatai

A telepítő külalaki adatainak beállításai (*Application Information, Main Window, Features*) egyetlen ablak különböző oldalain jelennek meg. Kattintsunk a ***Main Window*** fülre, hogy megadhassuk a telepítés bejelentkező ablakának paramétereit (17.4. ábra).

Itt annak a bizonyos, legtöbbször kék hátterű ablaknak a jellemzőit állíthatjuk be, mely mindvégig a telepítés alatt teljes képernyőre kivetítve fog megjelenni. A címének *(Main Title)* ne írjunk be túl hosszú szöveget, mivel ennek egyetlen sorban ki kell férnie. Ha pedig az ablak jobb-felső sarkában még egy képet is megjelenítünk (*Logo Bitmap*), akkor a címszövegnek már alig marad hely. A kép pozícióját és az ablak háttérszínét megváltoztathatjuk, mégis legtöbbször elfogadjuk az alapértékeket.
Figyelem! Az InstallShield Express csak a 16 színű bittérképeket kezeli; ezt a későbbi képbeállításoknál is figyelembe kell vennünk.



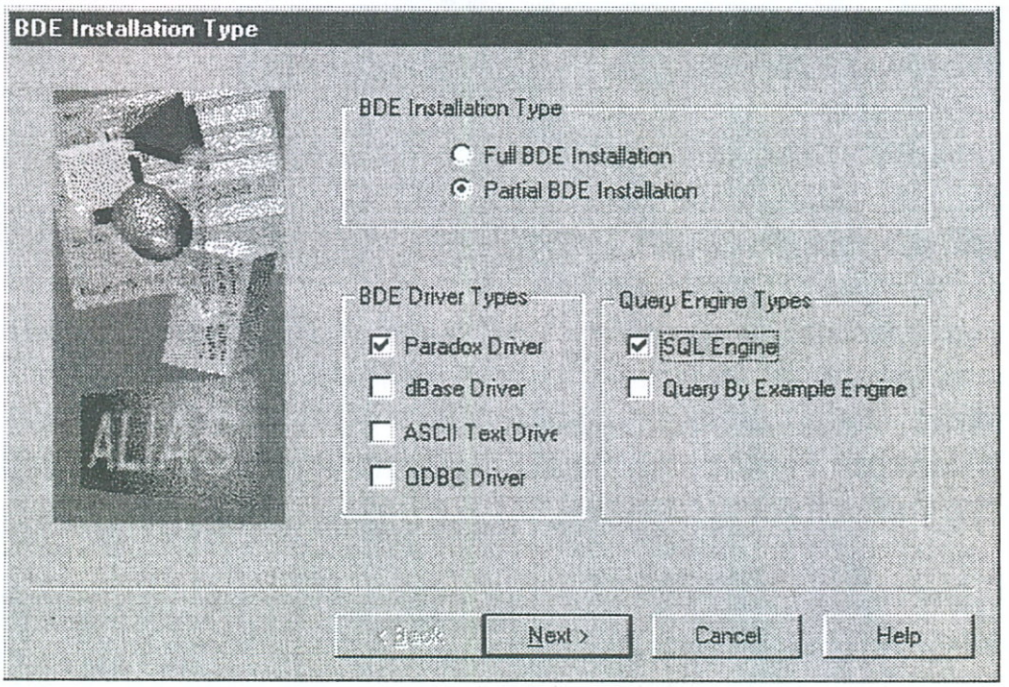
17.4. ábra. A telepítés ablakának paraméterei

A harmadik oldalon *(Features)* fogadjuk el a kipipált *Automatic Uninstaller* jelölőnégyzetet. Ha ezt tesszük, akkor a feltelepített alkalmazást kényelmesen le lehet majd törölni a célgépről minden feltelepített DLL állománnyal és regisztrációs adatbázis bejegyzéssel együtt. Ezt az alkalmazás programcsoportjában létező UNINST vagy a vezérlőpulton levő *Add/Remove Programs* programmal tehetjük majd meg.

17.4 A BDE állományainak kiválogatása

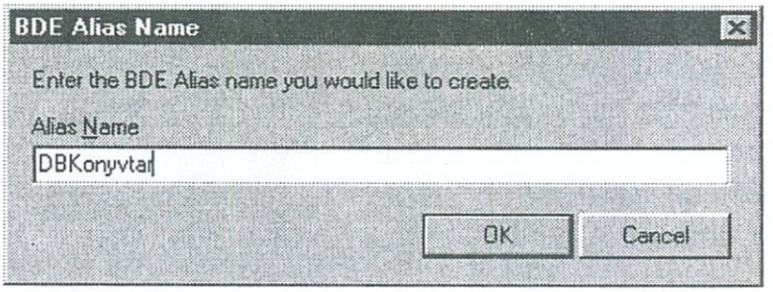
Ebben a pontban az alkalmazásunk által használt Delphi-specifikus részeket kell kiválasztanunk.

A **BDE** bizonyos részére mindenképpen szükség van. A Borland cég azt javasolja, hogy a teljes adatbázismotort telepítsük fel a célgépekre is. Ha nem fogadjuk meg tanácsukat, akkor lehetőség van a ténylegesen szükséges részek kiválogatására. Példánkban, mivel Paradox táblákkal dolgoztunk, a *Paradox drivert* és az *SQL Enginet* választottuk ki.



17.5. ábra. A BDE állományainak kiválogatása

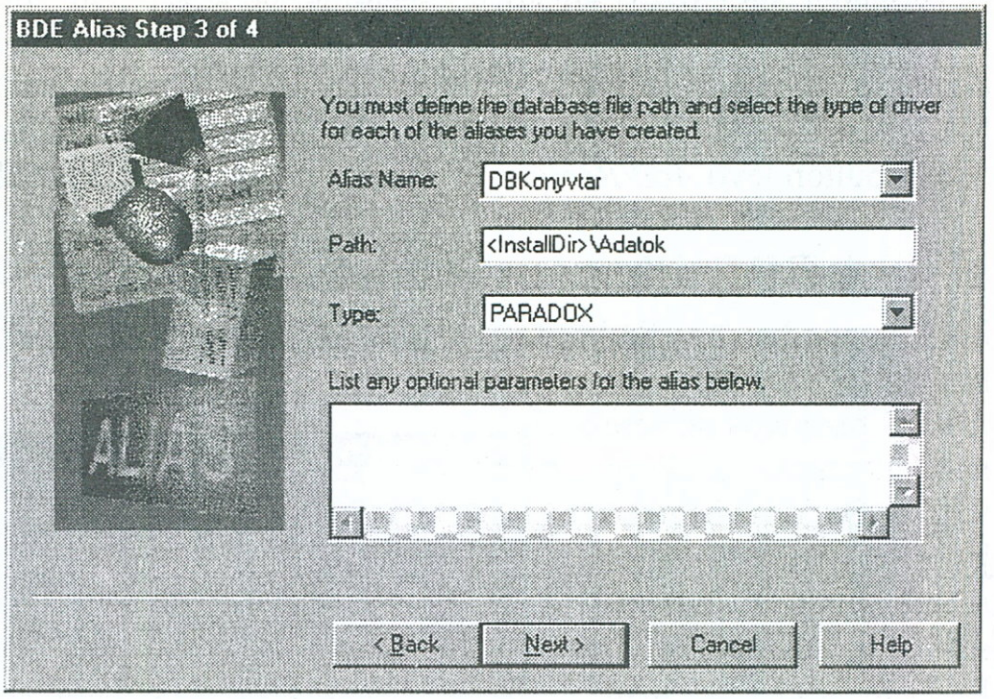
Lépjünk tovább a *Next* gomb segítségével. A következő lépésben az alkalmazás által használt **álnevet** kell megneveznünk (17.6. ábra).



17.6. ábra. Az álnév megadása

A következő ablak arra kérdez rá, hogy el szeretnénk-e menteni az álnévbeállításokat mind a Delphi 32-es, mind a Delphi 16-os konfigurációs állományába. Alkalmazásunk 32 bites, így az álnévbeállításokat az IDAPI32.CFG-ből olvassa majd ki. Nincs szükségünk a 16 bites álnevekre, ezért most ne pipáljuk ki a jelölőnégyzetet.

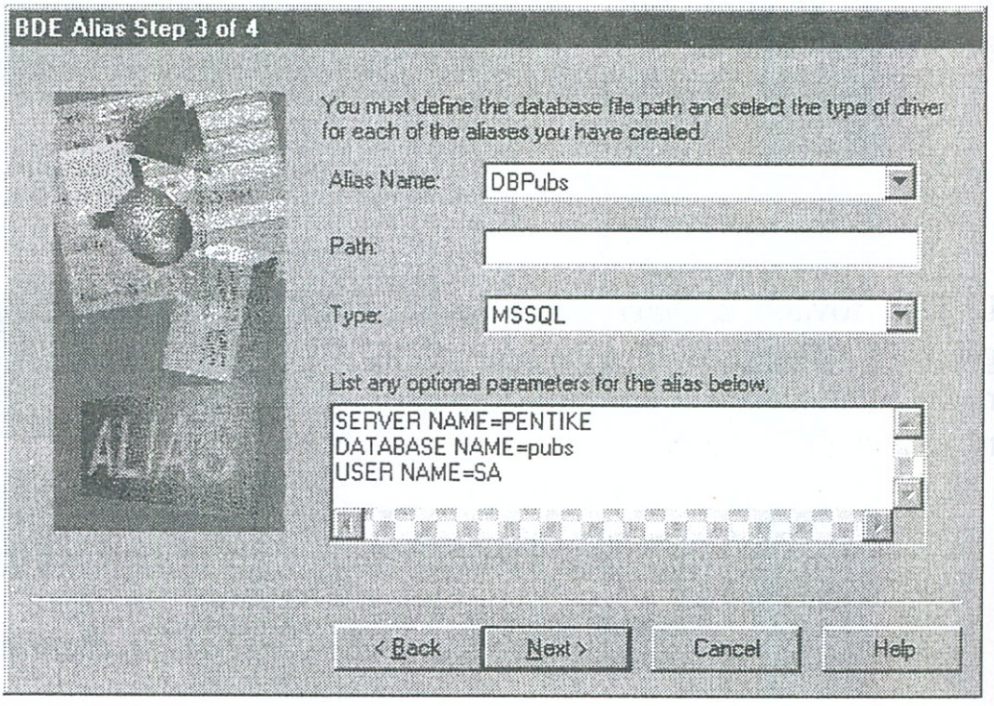
A következő lépésben **az álnév paramétereit** kell beállítanunk: adataink útvonalát és típusát.
Az útvonalnál használhatjuk a speciális InstallShield könyvtárazonosítókat: az *<InstallDir>* a telepítéskori célkönyvtárat jelenti, a *<WinDir>* a Windows könyvtárát, a *<WinSysDir>* a Windows SYSTEM könyvtárát... A teljes listát az *Add Groups and Files* pontban az új csoport hozzáadásánál megjelenő kombinált lista is tartalmazza (lásd a következő pontban).



17.7. ábra. Az álnévbeállítások

Példánkban Paradox táblákkal dolgozunk, melyeket majd a célkönyvtáron belül az Adatok könyvtárba fogunk telepíteni (17.7. ábra).

A *List any optional parameters for the alias below* szerkesztődobozban az álnév további beállításait írhatjuk. Erre az adatbázisszerverek esetén van szükség. Például, ha adataink a Microsoft SQL *pubs* adatbázisából származnának, akkor az álnév paramétereit az 17.8. ábra szerint kellene kitöltenünk.
Ilyenkor a *Path* doboz üresen marad.



17.8. ábra. A *DBPubs* a Pentike gépen található MSSQL *pubs* adatbázisnak az álneve

(Az MSSQL adatbázisok kezeléséhez szükség van a BDE-ből az SQL Engine részre, valamint szükség van még az SQL Links-ből az MSSQL állományaira.)

A következő párbeszédablakkal be is fejeződik a BDE konfigurálása. Ha adataink adatbázis-szerveren találhatók, akkor mindenképpen szükség van az *SQL Links* beállításra is. A párbeszédablak *Advanced* oldalán megtekinthetjük a kiválogatott állományok listáját. Ezeket a rendszer automatikusan elhelyezi a tömörítendő állományok csoportjába.

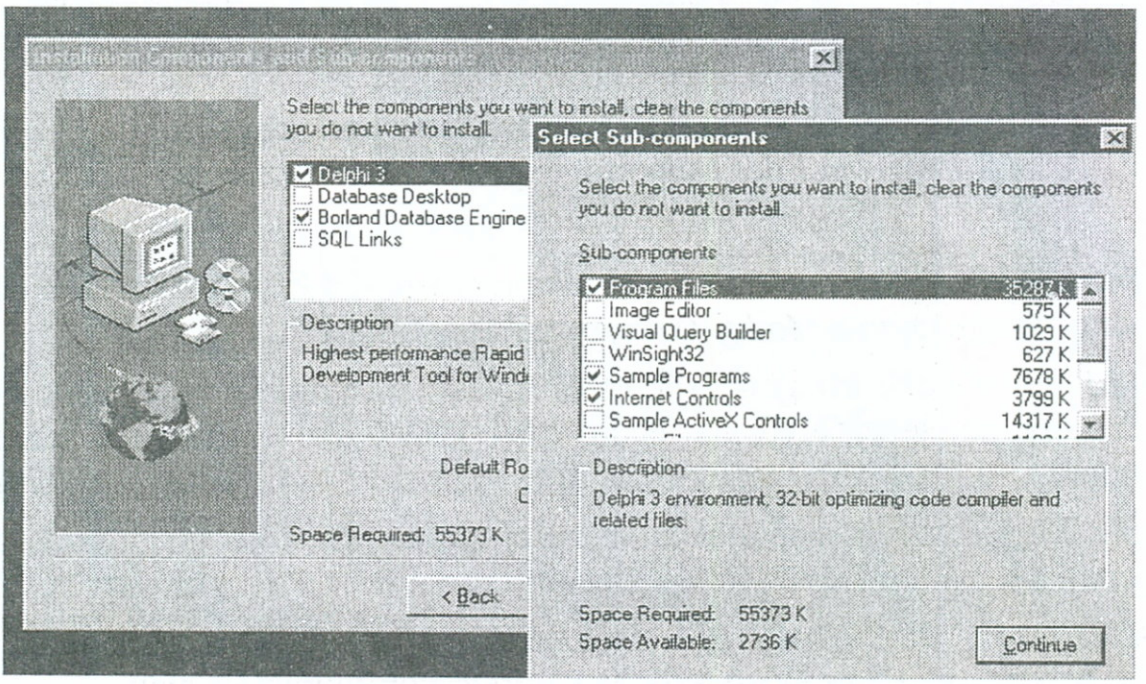
17.5 Az alkalmazás csoportjainak és állományainak megadása

Ebben a lépésben három fontos tennivalónk van:

- Meg kell adnunk az alkalmazás futásához szükséges állományokat. Ezeket különböző **csoportokba** fogjuk gyűjteni úgy, hogy az egy csoportba sorolt állományoknak azonos legyen a célkönyvtáruk. Lesz majd egy *Program* csoportunk, amibe az EXE állomány fog kerülni; lesz egy *Adatok* csoportunk a kezelt adatállományokkal stb. (lásd a 17.5.1. pontban)

Ha telepítőnkbe be szeretnénk építeni az egyéni telepítés lehetőségét is, akkor még két lépés áll előttünk:

- Létre kell hoznunk a **komponenseket**. A komponens egy fokkal nagyobb egység, mint az előző lépésben beállított csoport. Vegyük példának a Delphi telepítését (17.9. ábra): komponensnek számítana a *Delphi*, a *Database Desktop*, a *BDE* és az *SQL Links* csomag; ugyanakkor a *Delphi* komponens több részből áll: *Program Files*, *Image Editor*, *Sample Programs* stb. Ezeket nevezzük – InstallShield terminológiában – csoportoknak. (Bővebben lásd a 17.5.2. pontban.)

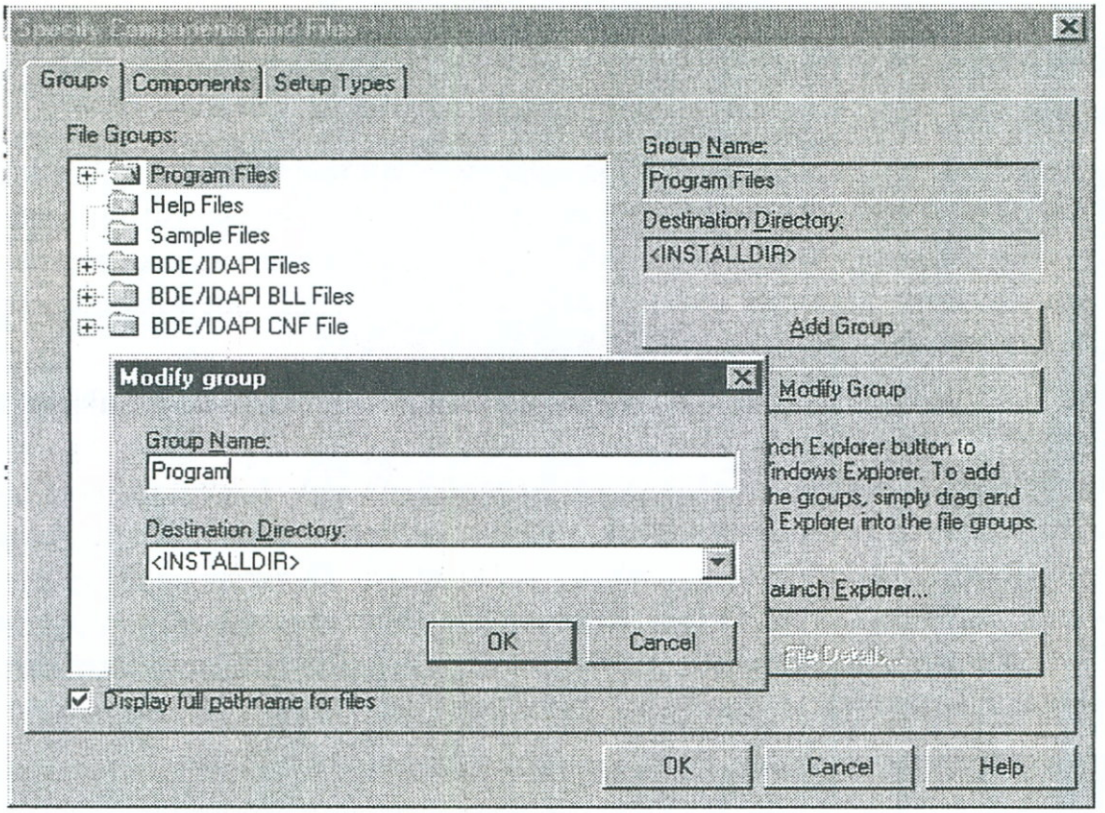


17.9. ábra. A Delphi telepítése közben kiválogatjuk a telepítendő komponenseket, és azon belül a csoportokat

- A harmadik lépésben **az általános, minimális és egyéni telepítési módokat** kell konfigurálnunk: mely komponensek szükségesek a minimális, melyek az egyéni, és melyek az általános telepítéshez (17.5.3. pont).

17.5.1 Az alkalmazás állományainak megadása

Kattintsunk a *Groups and Files* sorra a telepítőnk projektállományában. A megjelenő ablak első oldalán máris láthatók a csoportok (17.10. ábra):



17.10. ábra. Az alkalmazás állományait csoportokba soroljuk (A *Program Files* csoportot éppen most nevezzük át.)

- *Program Files*: ebben máris szerepel alkalmazásunk EXE állománya. Nevezzük át a csoportot *Program*-ra a *Modify Group* gomb segítségével.
- *Help Files*: a súgóállományok csoportja. Egyelőre még üres. Módosítsuk a csoport nevét *Súgó*-ra, az elérési útvonalát pedig *<InstallDir>*-re[1]. Az útvonal kombinált listájában láthatók a speciális InstallShield könyvtárazonosítók (*InstallDir, WinDir...*) .
- *Sample Files*: csoport a mintaalkalmazások számára. Mivel alkalmazásunkhoz nem tartoznak mintaállományok, töröljük le ezt a csoportot (*Delete* billentyű).
- A *BDE/IDAPI, BDE/IDAPI BLL és BDE/IDAPI CNF Files* csoportokat az InstallShield helyezte el ide az előző lépésben, a BDE konfigurációja alatt.

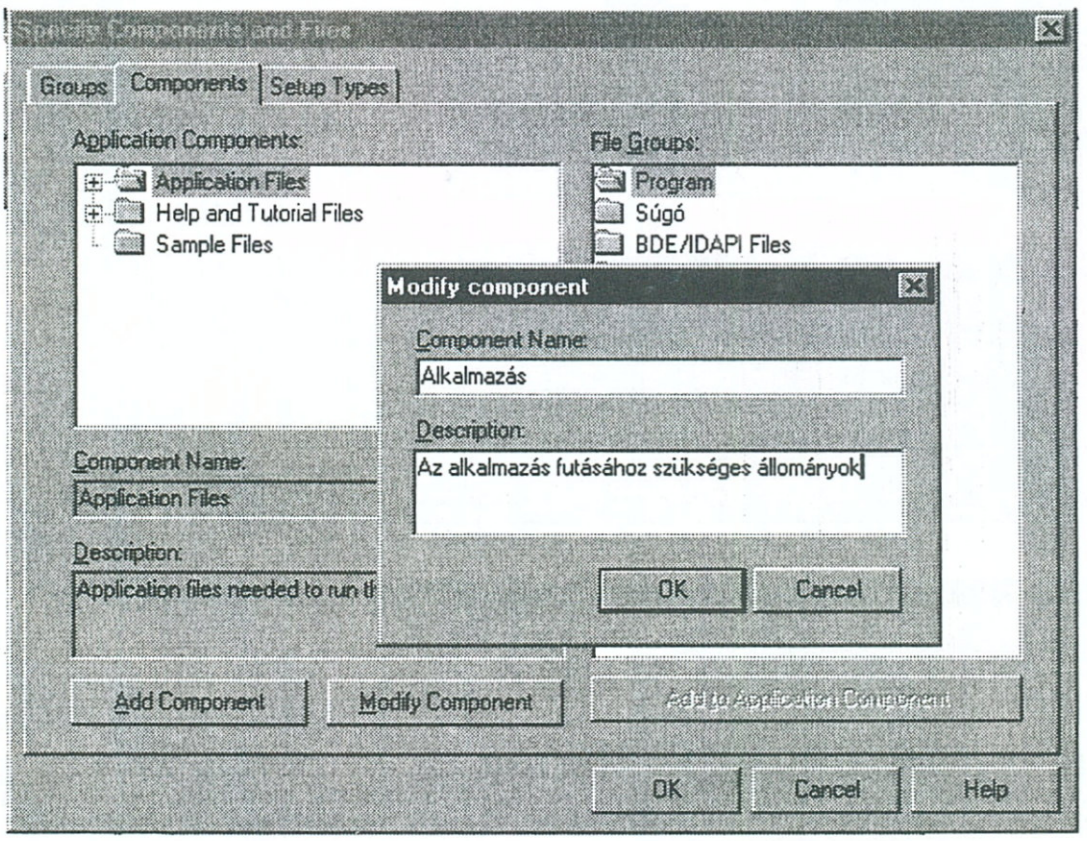
[1] Az alkalmazás a súgóállományt az *Application.HelpFile* jellemzőben megadott útvonalon fogja keresni. Mi annak idején azt írtuk, hogy *Application.HelpFile := 'KONYVTAR.HLP'*, ezért most a súgóállományt a telepítési célkönyvtárba kell tennünk, az EXE állomány mellé. (Természetesen meg lehetne az alkalmazásban is változtatni a hivatkozást, a lényeg csupán az, hogy az alkalmazásban és a telepítéskor beállítottak összhangban legyenek.)

Alkalmazásunknak szüksége van egy új csoportra az adatállományok számára. Hozzuk létre az *Adatok* csoportot az *Add Group* gomb segítségével. Elérési útvonala az *<InstallDir>\Adatok* lesz.

A következő lépésben az elkészített csoportokban elhelyezzük a megfelelő állományokat: indítsuk el a Windows Explorert (Intézőt), majd vonszoljuk sorban át az állományokat a megfelelő csoportokba: a 16_KONYVSUGO\KONYVTAR.HLP és KONYVTAR.CNT állományokat a *Súgó* csoportba, az ADATOK\ KONYVTAR*.* állományokat az *Adatok* csoportba. Ha alkalmazásunk használna saját DLL állományokat vagy futás idejű csomagokat (DPL), akkor azokat is most kellene áthelyezni a Program csoportba.

17.5.2 A komponensek konfigurálása

Lépjünk át a *Components* oldalra. Alkalmazásunkat két komponensre bonthatjuk: az *Alkalmazás* komponensre (benne az EXE, az adatok és a BDE), valamint a *Súgó* komponensre. Ezeket elvileg függetleníteni lehet egymástól: az alkalmazás működőképes súgó nélkül is, de adatállományok vagy BDE nélkül biztosan nem az.



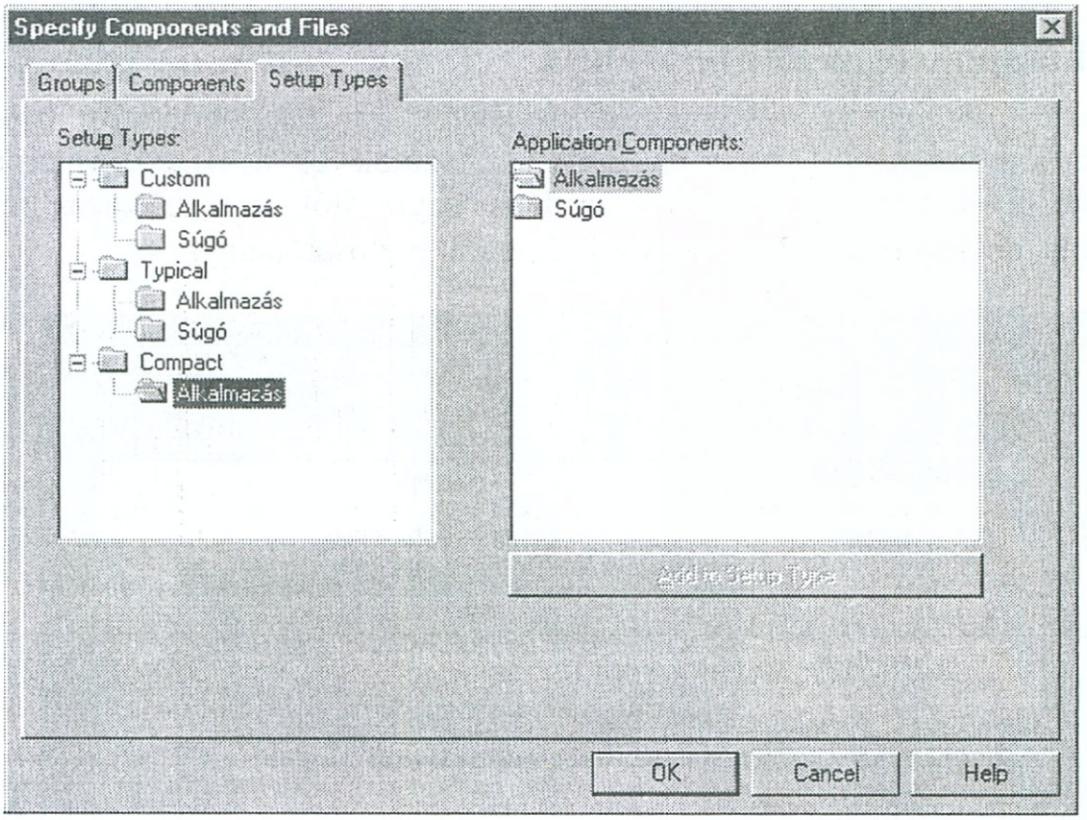
17.11. ábra. Alkalmazásunk komponensei

Nevezzük tehát át az *Application Files* komponenst *Alkalmazássá*, A *Help and Tutorial Files-t* pedig *Súgó*-ra. A *Sample Files* komponenst töröljük ki.
Ezután a jobb listadobozban látható csoportokat el kell helyeznünk egy-egy komponensben. Természetesen egy csoportot csak egy komponensben van értelme elhelyezni.

Példánkban a *Súgó* csoport a *Súgó* komponens része lesz, az összes többi csoport pedig az *Alkalmazás*-ban kap helyet.

17.5.3 Az általános, egyéni és minimális telepítés konfigurálása

Kattintsunk az utolsó, *Setup Types*, fülre. A jobb listában a komponenseket, míg a balban a telepítési módokat láthatjuk (17.12. ábra). Minden telepítési módnál mindkét komponens jelen van. Ez a beállítás megfelel az általános (*Typical*) és az egyéni (*Custom*) telepítésnek. A minimális (*Compact*) telepítésnél viszont elhagyható a súgó. Töröljük ezt ki a *Compact* telepítési módból.



17.12. ábra. Az általános, egyéni és minimális telepítés komponensei

17.6 A párbeszédablakok beállítása

Ebben a lépésben a telepítés párbeszédablakait fogjuk beállítani. Kattintsunk a *Dialog boxes* sorra a telepítőnk projektállományában. A megjelenő űrlap bal felében válasszuk ki a szükséges párbeszédablakokat.

Egyeseknél a kipipálásukon kívül még további adatokat is meg kell adnunk. Például a *User Information* párbeszédablaknál a *Settings* oldalon beállíthatjuk a felhasználótól bekérendő adatokat: név, cég és sorszám vagy csak név és cég.

Figyelem! A bittérképek itt is csak 16 színűek lehetnek. És még egy fontos információ: a *Preview* gombra megjelenő minta statikus, nem követi a beállításokat. Ezek csak telepítés közben lesznek láthatók.

17.7 A regisztrációs adatbázis bejegyzései

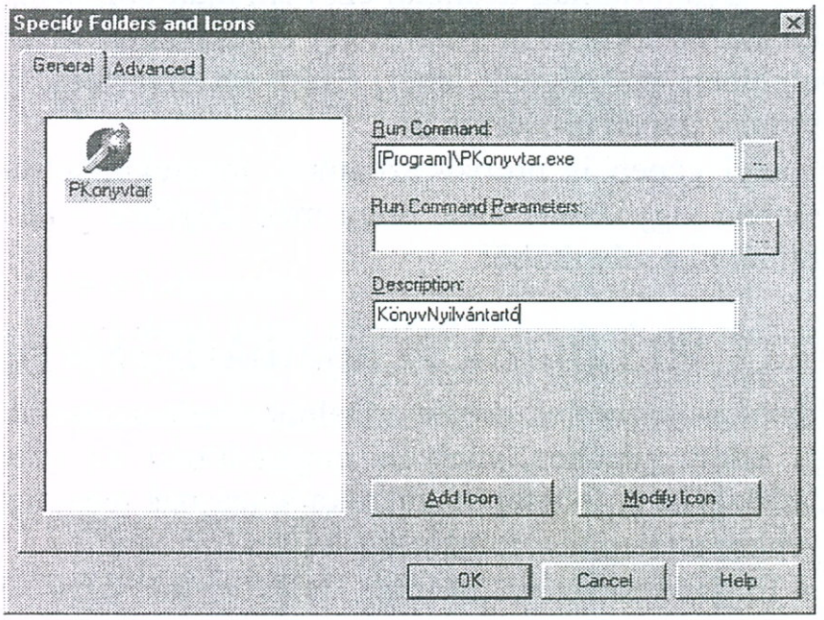
Ha alkalmazásunk saját regisztrációs bejegyzéseket használ, akkor ezeket a *Make Registry Changes* beállításban kell megadnunk. Erre legtöbbször nincs szükség, példánkban sincs, így továbbmehetünk.

17.8 A program csoportjának és ikonjának beállítása

Ebben a pontban a telepítendő alkalmazás indító parancsát, ikonját és csoportnevét állíthatjuk be.

A parancssort a rendszer „megelőlegezi", ezt fogadjuk el. Paraméterei nincsenek, viszont a leírását (az ikon alatti szöveget) írjuk át (17.13. ábra), majd kattintsunk a *Modify Icon* gombra.

Az *Advanced* oldalon az alkalmazás munkakönyvtárát, ikonját és az esetleges „forróbillentyű"-kombinációját állíthatjuk be.

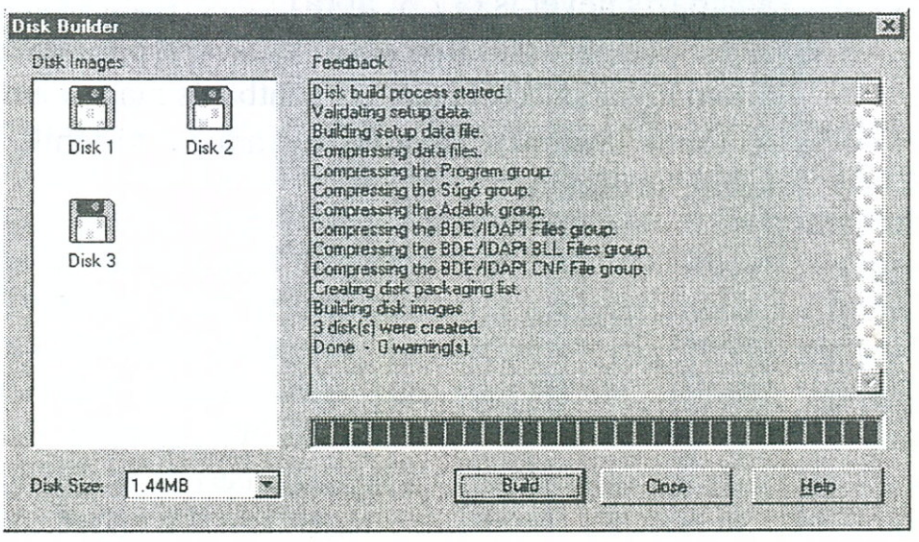


17.13. ábra. A program csoportjának és ikonjának beállítása

17.9 A telepítőkészlet létrehozása

Immár megadtunk minden fontos paramétert, következhet az állományok tömörítése, a telepítőkészlet elkészítése. Az állományokat ezután csak fel kell másolni a megfelelő lemezekre, és máris indítható lesz a telepítés.

Kattintsunk a projektállományban a *Disk Builder* sorra, adjuk meg a lemez méretét, majd indítsuk a tömörítést a *Build* gombbal. Hamarosan megjelennek a lemezek az űrlap bal felében. Tömörítés közben különböző figyelmeztető üzenetek (*warnings*) jelenhetnek meg, ezeket figyelmesen olvassuk el, és ha valamelyiket jogosnak érezzük, akkor intézkedjünk.



17.14. ábra. A tömörítés

Miután elkészültek a lemezek, mentsük le a projektállományt. Így később, az alkalmazáson végzett apróbb módosítások után újragenerálhatjuk a telepítőkészletet a most összeállított projektállomány alapján.

17.10 Próbatelepítés

A lemezek már készen állnak, elvileg kipróbálhatjuk a telepítést a saját gépünkön is (*Test Run*). Gyakorlatilag azonban ezt nem ajánlom, hiszen ha ezt tennénk, akkor a feltelepített alkalmazás törlésekor (*uninstall*) elvesznének olyan állományok is, melyekre később a Delphiben szükségünk lehet. Ezért azt javaslom, hogy a telepítést egy másik – Delphi nélküli – gépen próbáljuk ki.
A helyi gépen is próbálkozhatunk, megtekinthetjük a telepítéskori párbeszédablakokat, azonban vigyázzunk arra, hogy még idejében – az állománymásolások elkezdése előtt – lépjünk ki a telepítőből.

17.11 Mi változik az adatbázis-szerverek esetén?

Adatbázis-szerverek esetén a Delphi alkalmazásunkkal kliens oldalról közelítünk a hálózat egy adott gépéhez, melyen megtalálható az adatbázis-szerver program és maga az adatbázis is. A feltelepítendő alkalmazásnak tehát nem része maga az adatbázis.
Ilyenkor a telepítési folyamat a következő lesz:

- Először is vizsgáljuk meg, hogy az adatbázis létezik-e már vagy még nem. Ha még nem létezik, akkor nekünk kell létrehoznunk. Ezt legtöbbször az SQL szkriptállomány adatbázis-szerveren való lefuttatásával tesszük.
- Ha már létezik az adatbázis, akkor következhet a kliens alkalmazás telepítése. A telepítőkészlet elkészítésénél figyeljünk az alábbiakra:
  - A BDE állományain kívül most az SQL Links drivercsomagot is konfigurálnunk kell a megfelelő adatbázis-formátumtól függően.
  - Az álnév paramétereinél meg kell adnunk az adatbázis-szerver, és azon belül az adatbázis nevét is (17.8. ábra).
  - A csoportok összeállításánál csak a programállományok és a BDE kiválogatott állományai szükségesek, az adatbázis maga nem.
  - A többi lépésben minden ugyanúgy történik, mint a lokális adatállományok kezelése esetén.

18. Az alkalmazások közötti kommunikáció

A Windows rendszerekben egyszerre több alkalmazást is futtathatunk. A párhuzamosan futó alkalmazások egymással kapcsolatba is tudnak lépni: adatokat küldhetnek egymásnak, valamint felkérhetik egymást bizonyos tevékenységek elvégzésére. A kommunikációnak több megvalósítási módja is van, ezeket fogjuk áttanulmányozni ebben a fejezetben:

- Vágólap (*clipboard*)
- DDE (*Dynamic Data Exchange* = Dinamikus adatcsere)
- OLE (*Object Linking and Embedding* = Objektum csatolása és beágyazása)

18.1 A vágólap (clipboard) használata Delphiben

Mindnyájan helyeztünk már vágólapra szövegrészeket (akár a Delphi programból is). Lehet, hogy a vágólap tartalmát később egy másik Delphi programba helyeztük be, de az is lehet, hogy egy Word dokumentumba vittük át (így került a könyvbe a számtalan kódrészlet). A Delphi alkalmazás futáskori űrlapját szintén vágólapra helyezhetjük (*Alt + Print Scrn* billentyűkombinációval), majd ezt beilleszthetjük egy szövegszerkesztőbe. Kétségtelen tehát, hogy az alkalmazások közötti kommunikációnak egyik módja a vágólap.

Delphiben számos olyan komponens van, melynél már eleve megtalálhatók a *CopyToClipboard*, *CutToClipboard* és *PasteFromClipboard* metódusok. Ilyen például a *TEdit*, *TMemo*, *TDBImage*. Más komponensekbe, vagy akár alkalmazásainkba is beépíthetünk vágólap funkciókat a ***Clipbrd*** egységben deklarált ***Clipboard:TClipboard*** objektum segítségével. Ez egy előre deklarált objektum. Elég beszerkeszteni a programunkba a *Clipbrd* egységet, és máris használhatók a *Clipboard* jellemzői, metódusai. A vágólapon található információ formátumát a ***HasFormat*** metódusával kérdezhetjük le. Létezik néhány beépített formátum (*CF_TEXT* ⇨ szöveg, *CF_BITMAP* ⇨ bittérkép...), további formátumokat pedig a ***Formats*** jellemzőben helyezhetünk el.

Ha a vágólapon szöveg található, vagy oda szöveget szeretnénk elhelyezni, akkor az ***AsText*** jellemzőjét használjuk. Ha bittérképet vagy egyebet akarunk vágólapra tenni, vagy onnan elvenni, akkor az ***Assign*** metódust kell használnunk. Ezt szemlélteti az alábbi kódrészlet is:

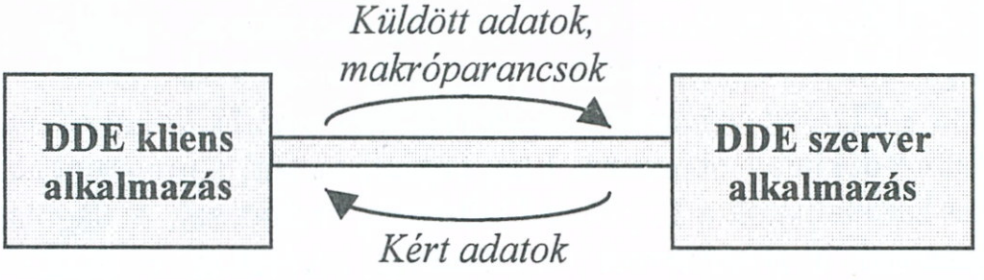
```
uses Clipbrd...;
...
{szöveges adat átmásolása az Edit1 szerkesztődobozból az Edit2-be}
Clipboard.AsText:= Edit1.Text;
...
If Clipboard.HasFormat(CF_TEXT) Then
  Edit2.Text:= Clipboard.AsText;
...
{kép másolása vágólapon keresztül}
ClipBoard.Assign(Bitmap1);
If Clipboard.HasFormat(CF_BITMAP) Then
  Bitmap2.Assign(Clipboard);
```

A Delphi programból vágólapra helyezett szöveget vagy képet természetesen más alkalmazásokba is beilleszthetjük. Ez fordítva is igaz: ha Wordből kimásolunk egy szót a vágólapra, akkor az a Delphi *Clipboard* objektumának *AsText* jellemzőjével lekérdezhető lesz.

Készítsen egy olyan listadoboz komponenst (neve legyen *TClipListBox*), mely rendelkezik a *CopyToClipboard*, *CutToClipboard* és *PasteFromClipboard* metódusokkal. Természetesen a vágólapra való másolás és helyezés az aktuálisan kijelölt elemre értendő, beillesztéskor pedig a vágólap tartalma a kijelölt elem elé kerüljön.

18.2 A DDE (*Dynamic Data Exchange*) technika

A dinamikus adatcsere (leginkább) szöveges információk cseréjét teszi lehetővé a párhuzamosan futó alkalmazások között. A beszélgető feleket itt **kliens-** és **szerver**-alkalmazásoknak nevezzük: a kliens a kezdeményező fél, a szerver pedig a „felkért", vagyis a kiszolgáló alkalmazás. Az adatcsere egy ún. **kommunikációs csatornán** keresztül történik a kliens alkalmazás kezdeményezésére (18.1. ábra).



18.1. ábra. A DDE csatornán keresztül beszélgető felek

Egy DDE kapcsolatban mindig a kliens kezdeményez: küldhet adatokat a szerverhez, és ezeket onnan vissza is kérdezheti (például egy Word dokumentum egy könyvjelzőjének[1] tartalmát, vagy egy Excel táblázat egy celláját...), de ugyanakkor ún. makróparancsokon (ezek szöveges utasítások) keresztül utasíthatja is a szervert bizonyos feladatok elvégzésére. Például, ha egy Delphi alkalmazásból programcsoportot és ikont szeretnénk létrehozni, akkor alkalmazásunknak DDE kapcsolatot kell létesítenie a programkezelővel. A csatorna létrejötte után az alkalmazásunknak két makróparancsot kell küldenie a programkezelő felé, ezzel előbb létrehozza a programcsoportot, majd a csoporton belül a programikont.
Kliensként más alkalmazásokkal is „beszélgethetünk" DDE-n keresztül, hogy pontosan melyekkel és hogyan, lásd az érintett program dokumentációjában. Ha a DDE csatornára már nincs többet szűkség, azt tételesen be kell zárnunk.

A DDE kommunikációs csatornának 3 eleme van, ezekkel írja le a kliens alkalmazás a cserélendő adatot:

- **Szolgáltatásnév (*DDEService*)**: a DDE szerver alkalmazás neve (általában ez az EXE neve kiterjesztés nélkül). Például a Wordnél a szolgáltatásnév „Winword", Excelnél „Excel". Egy alkalmazás pontos DDE neve megtalálható a dokumentációjában.
- **Téma (*Topic*)**: a cserélendő adatokat tartalmazó egység (legtöbbször állomány) neve. Például ALMA.DOC, TABLAZAT.XLS. A legtöbb alkalmazás rendelkezik egy speciális témával, a neve *System*. Ennek segítségével lekérdezhetjük, hogy egy alkalmazás az adott pillanatban milyen témakörökről „hajlandó beszélgetni" (például a Word lehetséges témái a *System*, valamint a nyitott dokumentumok nevei).
- **Elem (*Item*)**: az állományon belül a kapcsolt rész hivatkozása. Például egy Word dokumentumban a könyvjelzőnév, egy Excel munkalapon a cella hivatkozása (R1C2 vagy az Excel magyar változatában S1O2 = sor 1, oszlop 2).

Delphiben egyaránt készíthetünk DDE szerver és kliens alkalmazást a *System* palettán található komponensek segítségével: szerver alkalmazáshoz a *DDEServerConv* és *DDEServerItem*-re van szükség, kliens alkalmazáshoz pedig a *DDEClientConv* és *DDEClientItem*-re. A gyakorlatban Delphiben írt szerver alkalmazásra kisebb az igény, emiatt könyvünkben csak a DDE kliens alkalmazás készítését mutatjuk be:

18.2.1 DDE kliens alkalmazás készítése Delphiben

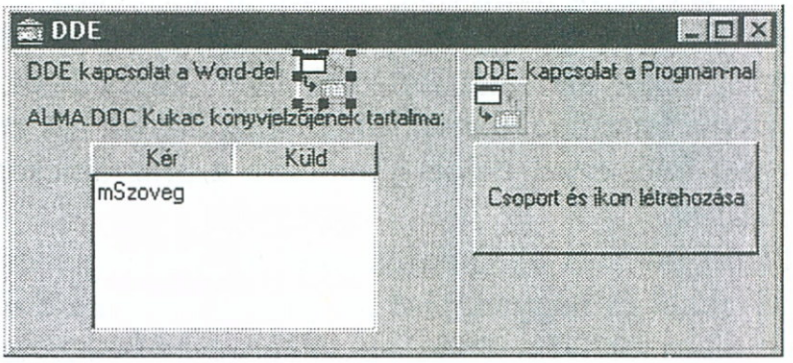
Készítsünk egy Delphi alkalmazást, mely DDE kapcsolatban áll a Worddel és a programkezelővel: a Worddel azért, hogy az ALMA.DOC állománybeli „kukac" könyvjelző értékét figyelje (olvassa és írja), a programkezelővel pedig azért, hogy makróparancsokkal egy programcsoportot és ikont hozzunk létre leendő programunk számára.

---

[1] Egy Word dokumentumban egy szövegrészt az ún. könyvjelzőkkel nevezhetünk meg; később a könyvjelző segítségével hivatkozhatunk a tartalmára.

Megoldás (🖫 Alkalmazás: 18_DDEKLIENS\DDEKLIENS.DPR, Dokumentum: ALMA.DOC)

Először tervezzük meg az alkalmazás űrlapját (18.2. ábra).
Bal részében a Word kapcsolatot követhetjük nyomon. A *Kér* gomb segítségével lekérdezzük, majd megjelenítjük az *mSzoveg* szerkesztődobozban az ALMA.DOC dokumentumbeli „kukac" könyvjelző értékét. A *Küld* gombra a szerkesztődoboz szövegét át fogjuk küldeni a Word dokumentumba, ezzel átírva a könyvjelző tartalmát.
A programcsoport és az ikon létrehozására vegyünk fel egy gombot az űrlap jobb részében.



18.2. ábra. A DDE kliens alkalmazásunk űrlapja

Alkalmazásunk két DDE szerver alkalmazással fog kapcsolatot kezdeményezni, ezért két *DDEClientConv* komponensre (*System* paletta) lesz szükségünk. Helyezzük el ezeket az űrlapon, nevük legyen *DDEClientConvWord* és *DDEClientConvProgman*. Jellemzőikkel, használatukkal most a konkrét példán keresztül fogunk megismerkedni.

18.2.1.1 DDE kapcsolat a Worddel

1. Először is indítsuk el a Word szövegszerkesztőt. Töltsük be az ALMA.DOC dokumentumot. Keressük meg benne a „kukac" könyvjelzőt: a *Szerkesztés/Ugrás* menüponttal ugorjunk a „kukac" könyvjelzőre. Az ugrás eredményeként kijelölt szövegrész nem más, mint a könyvjelző tartalma, ezt fogjuk a Delphi alkalmazásunkból lekérdezni és átírni.

2. Ha a DDE szerver alkalmazást készenlétbe helyeztük, akkor hozzuk létre a kapcsolatot. Ennek érdekében lépjünk vissza Delphibe, és állítsuk be a *DDEClientConvWord* komponens jellemzőit a következő táblázat szerint:

Komponens	Jellemző	Érték
DDEClientConvWord	DDEService	Winword
	DDETopic	ALMA.DOC

Mivel a *ConnectMode* jellemző értéke *ddeAutomatic*, a kapcsolat rögtön létrejön már a csatorna adatainak beállítása után. (Ha *ConnectMode* = *ddeManual*, akkor a kapcsolat inicializálása csak az *OpenLink* metódus hívásakor következik be.)

Azt is megtehetnénk, hogy a kommunikációs csatornát csak futási időben inicializáljuk a *SetLink ('Winword', 'Alma.Doc')* metódushívással. Ebben az esetben a kapcsolat

befejezéséért is mi vagyunk a felelősek; ezt a *CloseLink* metódussal valósíthatjuk meg.

3. Kódoljuk le a *Kér* és *Küld* gombok *OnClick* eseményjellemzőit. A könyvjelző értékét a *RequestData* metódussal kérdezzük le, és a *PokeData* metódussal állítjuk be. Figyelem, a DDE több alkalmazást érint, az ilyen műveleteknél pedig Windows szabvány szerint C típusú karakterláncok használatosak!

```
procedure TfrmDDE.btnKerClick(Sender: TObject);
begin
  {Lekérdezzük a könyvjelző értékét, a visszaadott C típusú
  karakterláncot Pascal karakterlánccá alakítjuk, és beírjuk az
  mSzoveg szerkesztődobozba.}
  mSzoveg.text:= StrPas(DDEClientConvWord.RequestData('kukac'));
end;

procedure TfrmDDE.btnKuldClick(Sender: TObject);
var p:PChar;
begin
  {Előállítjuk az átküldendő szöveget.}
  p:= StrAlloc(Length(mSzoveg.Text)+1);
  StrPCopy(p, mSzoveg.Text);

  {Felülírjuk vele a „kukac" könyvjelzőt.}
  DDEClientConvWord.PokeData('kukac',p);
  StrDispose(p);
end;
```

Tesztelje le az alkalmazást jelenlegi állapotában! Az adatcsere mindkét irányban a gombok hatására történik.

4. A klienshez érkező adat frissítése automatizálható egy *DDEClientItem* komponens segítéségével. Helyezzünk el egyet az űrlapon, és állítsuk be jellemzőit:

Komponens	Jellemző	Érték
DDEClientItemWord	DDEConv	DDEClientConvWord
	DDEItem	kukac

Ezzel tulajdonképpen a kommunikációs csatorna harmadik adatát adtuk meg: magát a könyvjelzőt. Ezek után a könyvjelző tartalmának minden egyes módosítása automatikusan be fog kerülni a *DDEClientItemWord* komponens *Text* jellemzőjébe. Mivel mi azt szeretnénk, hogy az új érték azonnal a szerkesztődobozban is megjelenjen, írjuk be a következő kódrészletet a *DDEClientItem.OnChange* eseményjellemzőjébe:

```
procedure TfrmDDE.DdeClientItemWordChange(Sender: TObject);
begin
  mSzoveg.Text:=DDEClientItemWord.Text;
end;
```

Tesztelje le az alkalmazást! A *Kér* gomb fölöslegessé vált, az adat már automatikusan megjelenik a szerkesztődobozban.

A *DDEClientItem* komponenst csak akkor használjuk kliens alkalmazásokban, amikor automatizálni szeretnénk az adatcserét.

18.2.1.2 DDE kapcsolat a programkezelővel

Ezúttal a programkezelővel létesítünk DDE kapcsolatot. A csatornán két makróparancsot fogunk elküldeni, ennek hatására a programkezelő létre fogja hozni a programcsoportot és az ikont.
Lépések:

1. Állítsuk be a *DDEClientConvProgman* komponens jellemzőit:

Komponens	Jellemző	Érték
DDEClientConvProgman	DDEService	Progman
	DDETopic	Progman

2. A következő lépés a *btnLetrehozas* lekódolása:

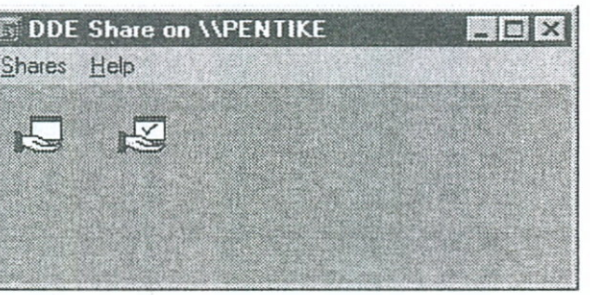
```
procedure TfrmDDE.btnLetrehozasClick(Sender: TObject);
var makro:PChar;
    S:String;
begin
  With DDEClientConvProgman Do
    begin
      {A programcsoport létrehozása}
      makro:= StrNew('[CreateGroup(CsopNév)]');
      ExecuteMacro(makro, False);
      StrDispose(makro);
      {A programikon létrehozása}
      S:='[AddItem('+ Application.ExeName + ', ProgramNév)]';
      makro:=StrAlloc(Length(S)+1);
      StrPCopy(makro,S);
      ExecuteMacro(makro, False);
      StrDispose(makro);
    end
end;
```

A programcsoportot a *[CreateGroup(CsopNév)]* makróparanccsal, a programikont pedig *[AddItem(ExeNév, ProgramIkonFelirat)]* utasítással hozzuk létre. A programkezelőt további makróparancsok végrehajtására is felkészítették, a pontos listát és paramétereket lásd a WIN32.HLP súgóállományban a *Shell Dynamic Data Exchange Interface* tartalomjegyzék bejegyzésben.

Figyelem! A súgóban leírtakkal ellentétben, a programkezelőben létező csoportok neveit nem a *Group* elem (*item*) lekérdezésével kaphatjuk meg, hanem a *Groups*-szal.

18.2.2 Hálózatos DDE kapcsolat (*NetDDE*)

DDE kapcsolatot a hálózat különböző gépein futó alkalmazások között is létesíthetünk. Például a saját gépünkről a hálózat egy másik gépén is létrehozhatunk programcsoportot és ikont. Természetesen ennek van egy előfeltétele: úgy, ahogyan egy másik gépen levő könyvtárat is csak akkor nyithatunk meg, ha azt valaki előzetesen kiajánlotta számunkra, DDE kapcsolatot is csak a célgépen kiajánlott DDE témakörökkel létesíthetünk. A témák kiajánlását a DDEShare programmal valósíthatjuk meg, természetesen a célgépre vonatkozóan (nem kötelező ugyanarról a gépről megtenni: *Shares/Select Computer...*). A kiajánlások felhasználófüggőek (*user*) lehetnek, azaz beállíthatjuk, hogy mely felhasználók és milyen jogokkal érhetik el az adott témát.



18.3. ábra. A *Pentike* gépen található DDE szerver alkalmazások kiajánlása

Feladat:
Készítsünk egy Delphi alkalmazást, mely lekérdezi a hálózat egy másik gépén található programcsoportokat.

Megoldás (🖫18_NETDDDE\NETDDE.DPR)

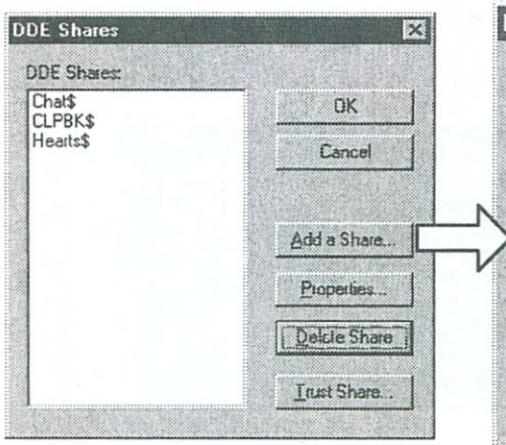
Lépések:
Először is tegyük elérhetővé a másik gépen található DDE szerver alkalmazást, példánkban a programkezelőt.

1. Ennek érdekében indítsuk el – a másik gépen – a DDESHARE programot (*Start/Run DDESHARE*).

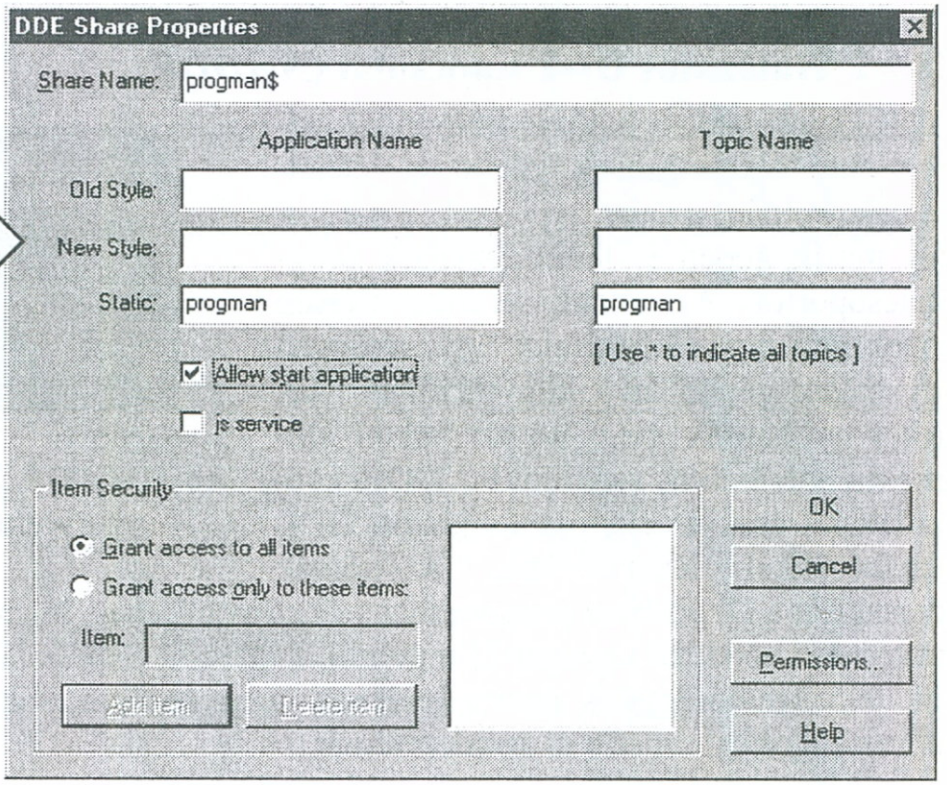
2. A két „kéz" (18.3. ábra) közül az elsőben az aktuális gép összes kiajánlott témáját láthatjuk (*global shares,* 18.4. ábra), míg a másodikban csak az aktuális felhasználó

csoportja által elérhetőket (*trusted shares*). Amikor egy témát ki szeretnénk ajánlani, előbb ezt fel kell vennünk a globálisak közé. Később innen az adott felhasználók átemelhetik a témakört a saját *trusted shares* témáik közé, ezzel beállítván azt, hogy ezt a hálózat bármely gépéről elérhessék.

Kattintsunk előbb az első „kézre". Az *Add a Share* gombbal ajánljuk ki a programkezelőt (18.5. ábra). A *Share Name* szerkesztődobozba a programkezelő kiajánlási nevét írjuk. A súgó szerint ennek ajánlatos '$'-ban végződnie. Az alatta található szövegdobozokat töltsük ki az alkalmazás, valamint a helyi témakör nevével. (Általában a csak "statikus" (*static*) üzemmódot használjuk.)

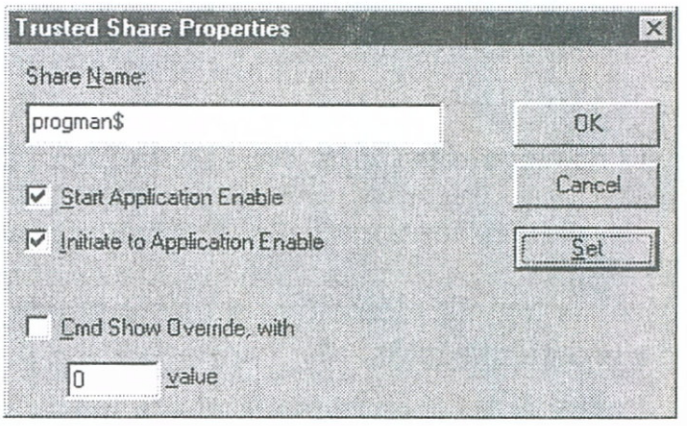


18.4. ábra. A pillanatnyilag kiajánlott DDE szerverek listája



18.5. ábra. A programkezelő kiajánlása

3. Kattintsunk a *Trust Share...* gombra. A megjelenő ablakban (18.6. ábra) pipáljuk ki a *Start Application Enable* és az *Initiate to Application Enable* jelölőnégyzeteket. Az elsővel azt állítjuk be, hogy egy DDE csatorna inicializálásakor a szerver alkalmazás automatikusan induljon el, ha még nem futott. A másodikkal nemcsak a pillanatnyilag létező, hanem az új DDE csatornák nyitását is engedélyezzük (ezt mindenképpen állítsuk be). Érvényesítsük a beállításokat a *Set* gombbal.

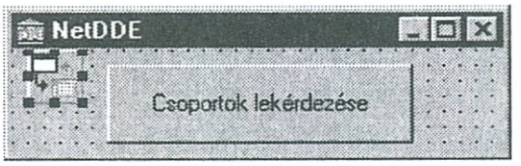


18.6. ábra. A *progman$* szerver-téma konfigurálása

A gyakorlat azt mutatja, hogy a *Set* gombot a globális beállítások után is célszerű meghívni, nem csak a felhasználószintű (*trusted*) beállításoknál.

4. Ha most megnyitjuk a *trusted shares* listánkat (második „kéz"), akkor benne lesz a *progman$* téma is. Ez azt jelenti, hogy a hálózat más gépeiről is elérhetjük szolgáltatásait.

5. Menjünk vissza a kliens gépre, és hozzuk létre a Delphi alkalmazást (18.7. ábra). A *TDDEClientConv* komponens jellemzőit most a következőkre kell beállítanunk:



18.7. ábra. NetDDEKliens alkalmazásunk

Komponens	Jellemző	Érték
DDEClientConv	DDEService	*gépnév*\ndde$
	DDETopic	progman$

6. A csoportok lekérdezését a következő kódrészlettel valósítjuk meg:

```
procedure TfrmNetDDE.btnLekerdezesClick(Sender: TObject);
begin
  ShowMessage(StrPas(DDEClientConv.RequestData('Groups')));
end;
```

Tesztelje le az alkalmazást! Hozzon létre a másik gépen egy új csoportot és benne ikonokat (természetesen a programikonok a másik gépen található programokra hivatkozzanak)!

A DDEShare a Windows NT-hez tartozik, a Windows 95 és Windows for Workgroups esetén ez a segédprogram a megfelelő Resource Kit[1]-ben található.
További adatokat a NetDDE szolgáltatásról, valamint a DDEShare programról Microsoft-forrásokban találhatunk (például: *http://support.microsoft.com, Resource Kit, TechNet*[2]...).

18.3 Az OLE (*Object Linking and Embedding*) technika

Az OLE (objektumok csatolása és beágyazása) az alkalmazások közötti adatcsere és kommunikáció legfejlettebb formája. Segítségével programjainkat olyan szolgáltatásokkal is kibővíthetjük, melyeket más, létező alkalmazások bevonásával fog ellátni. A DDE-vel szemben itt nem két teljesen független alkalmazás között folyik az adatcsere és az utasítá-

[1] Ez egy segédprogram-csomag, melyet a Microsoft CD-n terjeszt a Windows rendszerekhez kiegészítésül.

[2] A TechNet a Microsoft által havonta CD-n terjesztett főleg kiegészítő dokumentációkat tartalmazó csomag.

sok küldése, hanem egy alkalmazás (*konténer*) szerves részévé tudunk tenni olyan elemeket és ezek szolgáltatásait (*objektumokat*), amelyek csak egy másik alkalmazás képes kezelni (*szerver*). Például egy Word dokumentumba beágyazott Paintbrush kép is csak a Paintbrush bevonásával szerkeszthető. Delphi alkalmazásainkba sem kell például szövegszerkesztő vagy táblázatkezelési funkciókat beépítenünk; alkalmazásunk beágyazva vagy csatolva tartalmazhat egy szöveget, vagy egy táblázatot, és valahányszor ezt szerkeszteni, nyomtatni... akarnánk, segítségül hívhatjuk a Word vagy Excel programokat.
A kapcsolatban álló feleket itt konténer és szerver alkalmazásoknak nevezzük: a **konténer** tartalmazza a beágyazott vagy csatolt objektumot (szöveget, képet, táblázatot...), a **szerver** pedig maga az objektumot kezelni tudó alkalmazás.

18.3.1 OLE 1.0, OLE 2.0, OLE automatizmus

Az OLE technikának több változatát ismerjük:

- **OLE 1.0**:

  Az OLE 1.0 technika szerint működő OLE szerver alkalmazás egy teljesen különálló ablakban jelenik meg; ebben megszerkeszthetjük az objektumot, majd a *Frissítés*, *Kilépés* menüpontokkal visszaléphetünk az OLE konténer alkalmazásba. Ilyen OLE szerverek például a Word 6.0 és az Excel 4.0.

- **OLE 2.0**:

  Ezek a szerverek már helyben, az OLE konténer alkalmazás ablakában jelennek meg. Ilyenkor a két alkalmazás menüpontjai összefésülődnek a menüpontok *GroupIndex* értékének növekvő sorrendjében. Ugyanakkor vannak olyan menüpontok is, melyek „eltűnnek": a konténer 1, 3 és 5 *GroupIndex* értékű főmenüpontjainak helyére a szerver megfelelő menüpontjai kerülnek. Ez teszi lehetővé azt, hogy a konténer alkalmazás *Szerkesztés, Formátum* és más objektum-specifikus menüpontjának helyét az OLE szerver szerkesztési, formázási menüpontjai vegyék át. Az Office 95 és 97 programok már OLE 2.0 technika szerint működnek.

- **OLE automatizmus**:

  Az OLE automatizmus az OLE 2.0 lehetősége, mely lehetővé teszi a szerver alkalmazás funkcióinak kihasználását anélkül, hogy ez bármilyen ablakban megjelenne. Delphiből kinyomtathatunk például egy Word dokumentumot anélkül, hogy bármit is észlelnék ebből a képernyőn. A merevlemez kerregése ugyan sejtet valamit, de a Word nem fog ablakban megjelenni. Ilyenkor az OLE szerver alkalmazás csak ideiglenesen (a funkció ellátásának idejére) töltődik be a memóriába (*process*-ként és nem *task*-ként). Az OLE automatizmusban résztvevő alkalmazásokat szokás még *OLE Automation Controller*-nek és *OLE Automation Server*-nek is nevezni.

Az OLE technika erőforrásigénye nagyobb, mint a DDE vagy vágólapos adatcseréé, ezért gondoljuk meg, mikor és milyen gépen alkalmazzunk.

Delphiben készíthetünk OLE konténer, OLE automatizmus kontroller, valamint OLE automatizmus szerver alkalmazást is. A szerverek ActiveX, OCX vezérlőelemek formájában hozhatók létre.

Az OLE konténer alkalmazásokban a *TOLEContainer* komponens segítségével ágyazhatunk be, illetve csatolhatunk egy objektumot. Csak el kell helyezünk az űrlapon egy ilyen komponenst, kattintunk rá duplán, és innen kezdve már, gondolom, mindenkinek ismerős a kép: ugyanúgy választhatunk a telepített OLE szerverek között, mint ahogyan ezt Wordben is megtehetjük a *Beszúrás/Objektum* menüpont hatására.

Az OLE automatizmus már egy kicsit izgalmasabb feladat. Tulajdonképpen itt arról van szó, hogy a kontroller alkalmazás ideiglenesen létrehoz egy szerver típusú objektumot, meghívja ennek szükséges metódusait (feladattól függően), majd a munka végeztével megszünteti ezt. Ahhoz, hogy ez általánosan alkalmazható legyen, az objektumváltozót *Variant* típusúnak kell deklarálnunk. Ismernünk kell a hívott OLE szerver objektumait, jellemzőit, metódusait, valamint a metódusok paraméterezését. Az objektum létrehozása a *CreateOLEObject(SzerverOsztályNév)* metódussal történik.
Nézzünk erre egy példát:

18.3.2 OLE automatizmus Delphiben

Küldjünk körlevelet egy Delphi adathalmazban szereplő személyeknek. A körlevél szövege a KORLEVEL.DOC állományban található. Ez egy Word körlevél típusú dokumentum, mely az adatait az ADATOK.TXT-ből olvassa ki. A feladatunk tehát az, hogy a címzendő személyek adatait írjuk át ebbe az állományba, majd fésüljük össze a körlevél szövegét az adatokkal. Nyomtassuk ki az így létrehozott leveleket.

Megoldás (🖫 Alkalmazás: 18_OLEAUT\LEVEL.DPR
A levél szövege: 18_OLEAUT\KORLEVEL.DOC
Az levél adatállománya: 18_OLEAUT\ADATOK.TXT)

Hozzunk létre egy új Delphi alkalmazást. Legyen benne egy adatmodul egy táblakomponenssel (*tblSzemely*). Ezt most az egyszerűség kedvéért irányítsuk a DBKonyvtar álnév által mutatott IRO.DB állományra (most íróinknak küldjük a körleveleket). Helyezzünk el az alkalmazás űrlapján egy gombot, *OnClick* eseményébe pedig a következőket írjuk:

```
procedure TForm1.btnKorlevelClick(Sender: TObject);
var  F:TextFile;
     WordApp, MyDoc:Variant;
begin
  Screen.Cursor:= crHourGlass;
  {Az ADATOK.TXT állomány feltöltése a személyek adataival}
  AssignFile(F,'c:\adatok.txt');
  Rewrite(F);
  With DM.tblSzemely Do
    begin
      First;
      {Oszlopfejlécek Tab-bal elválasztva}
```

```
      Writeln(F,'Azonosito',#9,'Nev');
      {A tényleges adatok}
      While Not EOF Do
        begin
          Writeln(F,Fieldbyname('IroAz').AsString,#9,
                    Fieldbyname('Nev').AsString);
          Next;
        end;
    end;
  CloseFile(F);

  Try
    Try
      WordApp:= CreateOleObject('Word.Application');
      MyDoc:=WordApp.Documents.Open('c:\korlevel.doc');
      {Az összefésülés egy új állományba történjen (wdSendToNewDocument=0)}
      MyDoc.MailMerge.Destination:= 0;
      {most következik az összefésülés}
      MyDoc.MailMerge.Execute;
      {Az aktív dokumentum (a kész körlevelek) kinyomtatása}
      WordApp.ActiveDocument.PrintOut(False);
      {False => Nem engedi tovább a Wordöt a nyomtatás befejezéséig}
    Except
      ShowMessage('Valami gond van a Worddel!');
    End;
  Finally
    If Not VarIsEmpty(WordApp) Then
      {Kilépünk a Wordből úgy, hogy nem mentjük a változtatásokat}
      WordApp.Quit(0);
    Screen.Cursor:= crDefault;
  End;
end;
```

A Winword *process*-ként van jelen a memóriában a *WordApp* inicializálásától egészen a *WordApp.Quit*-ig. Állítólag a *process-nek* a *WordApp* változó megszűnésével (az eljárás végén) automatikusan el kellene tűnnie, a valóság viszont az, hogy a módosított és le nem mentett dokumentumok meggátolják ebben a Wordöt. Emiatt célszerű minden szerver alkalmazást tételesen bezárni a *Quit* metódussal, mielőtt még a kliens eljárás véget érne – "biztos, ami biztos" alapon.

A következő fejezetben egy több rétegű (*multi-tier*) alkalmazást készítünk. Amint látni fogjuk, ezt is speciális, OLE (pontosabban COM) elven működő komponensek teszik lehetővé.

19. Több rétegű (*multi-tier*) adatbázis-kezelés

Amint erről már a 7. fejezetben is szó esett, a több rétegű alkalmazások legalább három rétegből állnak, és ezek a részek külön-külön gépeken futhatnak. Általában a következő az eloszlás: az adatbázis tárolása és közvetlen kezelése az adatbázis-szerveren történik; az alkalmazás-logika egy középső rétegben (*middle-tier*) található; az egyes gépekre pedig csak az ún. „sovány" (*thin*) kliens kerül, mely azért sovány, mert csak a felhasználói felületet tartalmazza. Ez Delphiben azt is jelenti, hogy a kliens csak egy EXE állományból áll, már nem igényli a BDE jelenlétét, hiszen most az adatokat a középső réteg szolgáltatja számára. Csak a középső rétegben van szükség a BDE-re és az SQL Links csomagra, mivel itt történik az adatok „kibányászása" az adatbázis-szervertől. Mivel ez a réteg adatokat szolgáltat, ***Data Broker***-nek is nevezik.

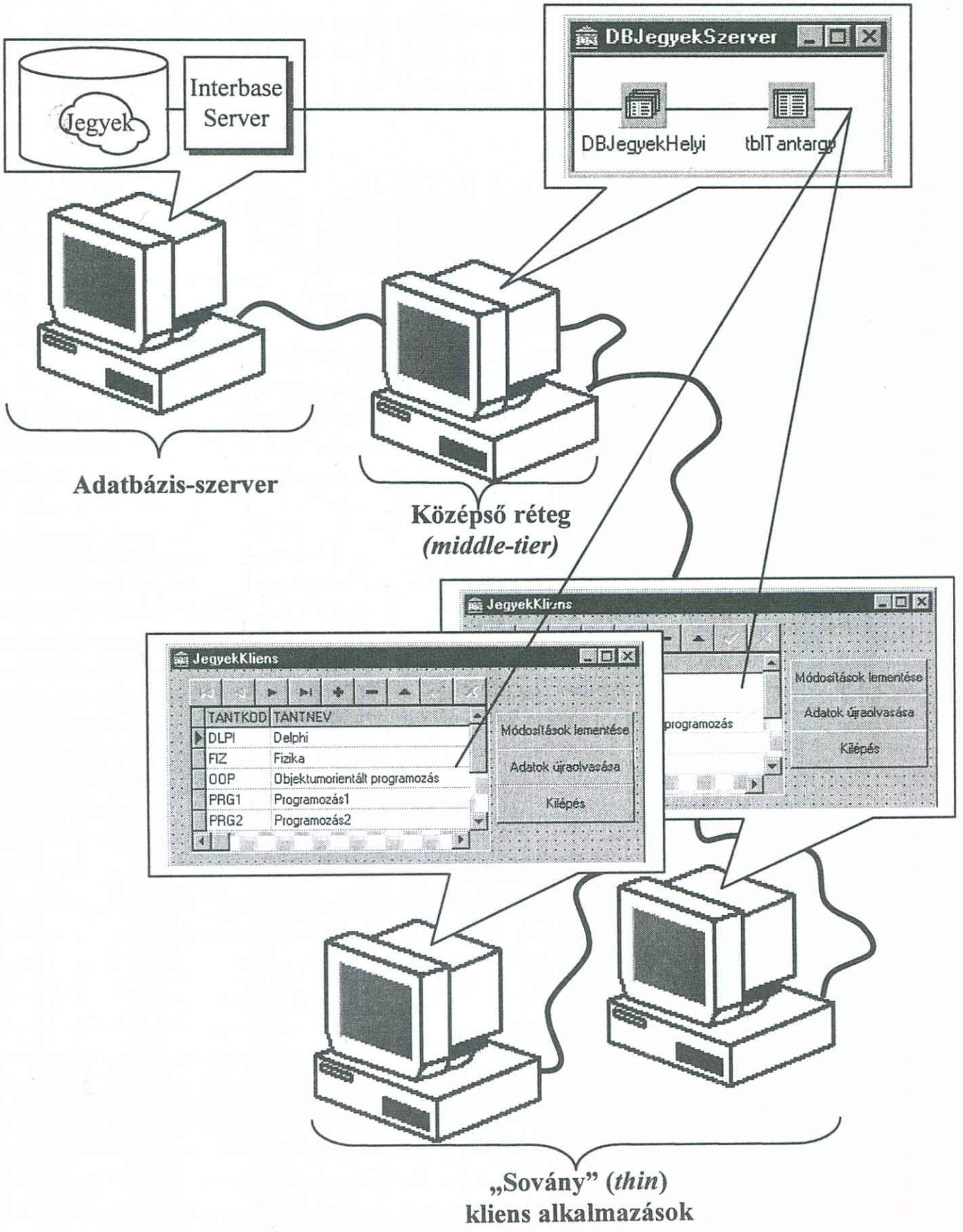
Ebben a fejezetben egy egyszerű, de a legfontosabb részeket tartalmazó három rétegű alkalmazást fogunk készíteni. Adatszolgáltatóként az Interbase SQL Server for Windows NT/95 adatbázis-szervert használjuk, a középső réteget és a kliens alkalmazást pedig Delphiben fejlesztjük ki. Ezt az alkalmazást csak a Delphi 3 kliens/szerver verziójában készíthetjük el, csak ez támogatja a több rétegű alkalmazások készítését.

19.1 Feladatspecifikáció

Készítsünk egy három rétegű alkalmazást. Természetesen ebből a három rétegből csak a középsőt és a klienst készítjük el Delphiben, adatkezelő rétegként az Interbase adatbázis-szervert használjuk. A középső rétegű alkalmazásnak az lesz a feladata, hogy a JEGYEK.GDB adatbázis *Tantargy* táblájának adatait elérhetővé tegye a nagyvilág számára, azaz *Data Broker*-ként működjön. A kliens alkalmazás a középső rétegből szolgáltatott *Tantargy* tábla adatait fogja megjeleníteni, szerkeszteni. Természetesen több klienst is elindíthatunk majd párhuzamosan egy, vagy akár több gépen is, ezek mindnyájan a középső rétegű adatszolgáltatóhoz csatlakoznak.

Megoldás (🖫 Adatbázis: ADATOK\JEGYEK\JEGYEK.GDB
Középső réteg: 19_MULTITIER\JEGYEKSZERVER.DPR,
Kliens: 19_MULTITIER\JEGYEKKLIENS.DPR)

Az adatbázist már a 14. fejezetben létrehoztuk. Most a feladatot két lépésben fogjuk megoldani: előbb elkészítjük a középső réteget, majd a kliens alkalmazást. A teljes alkalmazás szerkezetét a 19.1. ábra szemlélteti. Természetesen ennek nem kell mindenképpen 3-4 gépen lennie. Mi most mindezt egyetlen gépen visszük véghez.



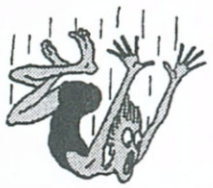
19.1. ábra. Három rétegű alkalmazásunk szerkezete

19.2 A középső réteg elkészítése

A középső réteg adatszolgáltatóként működik, így lényegében egyetlen speciális adatmodulból áll, mely lehetővé teszi, hogy a kliens alkalmazások az általa szolgáltatott adatokat el tudják érni. Lesz rajta egy adatbázis-komponens, és annyi tábla-, lekérdezés-... komponens, ahányra a feladatban szükség van. Most csak a tantárgyak adataira vagyunk kíváncsiak, ezért egy *TTable*-t fogunk csak benne elhelyezni.
Lépések:

1. Először is meg kell adnunk az adatok helyes és **teljes** elérési útvonalát, mivel a középső rétegű alkalmazás és az adatbázis-szerver elvileg más és más gépeken helyezkedhetnek el. Ennek érdekében hívjuk meg a Database Explorer programot, és a *DBJegyek* álnév *Server Name* jellemzőjét egészítsük ki az adatbázis-szerver gépének nevével[1]. Nálam ez *Pentike*.

   ```
   Server Name = \\Pentike\c:\Adatok\Jegyek.GDB
   ```

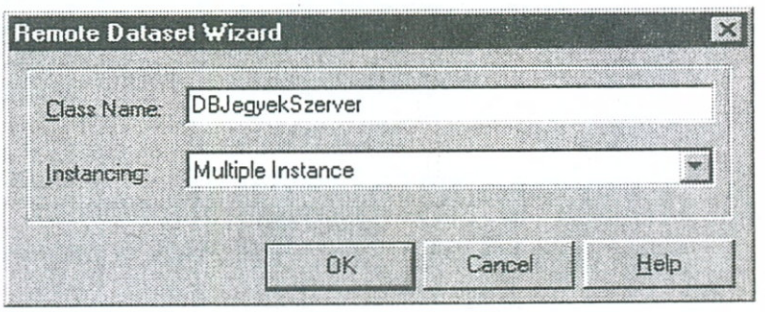
   Mint láthatjuk, ez a hivatkozás nem azonos a Microsoft hálózatokban használt UNC[2]-vel (\\gép\erőforrás). Mi itt a gép neve helyett az Interbase szerver nevét írtuk (\\Pentike), az erőforrás kiajánlási neve helyett pedig az adatbázis lokális elérési útvonalát (c:\...). Az elérési útvonalat maga az adatbázis-szerver kezeli le, lokálisan. Az adatbázis (.GDB) kiajánlása fájlszinten fölösleges, sőt káros is biztonsági szempontból.

   Mentsük le az álnév módosításait a *File/Apply* menüpont segítségével. Bizonyosodjunk meg az útvonal helyességéről úgy, hogy duplán kattintunk az álnévre: ha az útvonal megfelel a valóságnak, akkor a jelszó megadása után máris látható az adatbázis tartalma. Természetesen, ehhez az Interbase Servernek is futnia kell.

2. Lépjünk be a Delphibe, és hozzunk létre egy új alkalmazást. Ebben egy speciális adatmodult fogunk létrehozni.

3. Hívjuk meg a *File/New...* menüpontot, majd válasszuk a *Remote Data Module* opciót. A megjelenő párbeszédablakba a leendő adatmodul nevét kell beírnunk. Ezen a néven fogják a kliens alkalmazások is látni.

   Töltsük ki az adatokat a 19.2. ábra alapján.



19.2. ábra. Távoli adatmodul létrehozása

---

[1] Egy gép neve megtekinthető az asztalon (*desktop*) levő *Network Neighborhood* gyorsmenüjéből a *Properties* megjelenítésével.

[2] **UNC** = **U**niversal **N**aming **C**onvention. Ezt a hivatkozási rendszert használjuk a Windows alapú hálózatokban.

Az adatmodulunk tulajdonképpen egy COM[1] szerver objektum, melyet most a varázslóval hoztunk létre, és az alkalmazás futtatásával fogunk regisztrálni. A kliens alkalmazások ezt fogják használni, futáskor „berántják" a memóriába, akárcsak az OLE technikában az OLE konténer az OLE szerver alkalmazást. Az *Instancing* beállítással a COM objektum működését befolyásolhatjuk: például a *Multi Instance* hatására minden kliens egy közös adatszolgáltatót fog használni. Ha viszont *Single Instance* értéket állítunk be, akkor minden kliens saját szerver (adatkiszolgáló) objektumot indít el.

Az adatszolgáltatónak egy interfész része is van, az itt definiált rutinokat hívhatják a kliens alkalmazások. Vessünk egy pillantást a legenerált kódra:

```
unit UDMJegyek;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, ComServ, ComObj, VCLCom,
  StdVcl, BdeProv, DataBkr, JegyekSzerver_TLB, Db,
  DBTables; {ezeket a varázsló építette be}

type
  TDBJegyekSzerver = class(TDataModule, IDBJegyekSzerver)
  {Az IDBJegyekSzerver az adatkiszolgálónk interfész
  osztálya. Ezt is varázsló hozta létre automatikusan.
  Helye a JegyekSzerver_TLB.PAS állományban van.}
 ...
  end;
...
initialization
  {Ez a kódrészlet az alkalmazás futásakor fog végre-
   hajtódni, ez végzi el a COM objektum regisztrálását.}
  TComponentFactory.Create(ComServer, TDBJegyekSzerver,
                  Class_DBJegyekSzerver, ciMultiInstance);
end.
```

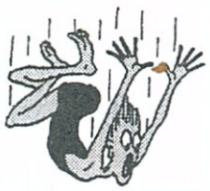
4. Mentsük le alkalmazásunk állományait: az adatmodul⇨UDMSZERVER.PAS, az űrlap egysége⇨USZERVER.PAS, a projektállomány⇨JEGYEKSZERVER.DPR.
5. Helyezzünk el az adatmodulon egy *TDatabase* és egy *TTable* komponenst. Állítsuk be adataikat a táblázatnak megfelelően:

---

[1] COM = ***C**omponent **O**bject **M**odel*. Ez egy olyan szabvány, mely előírja, hogy a megosztottan használható objektumoknak milyen interfésszel (metódusokkal) kell rendelkezniük. Ennek egy sajátos esete az OLE technika, amit az előző fejezetben ismerhettünk meg. Az OLE szerver objektumok – a Word, az Excel, de a Delphiben fejlesztett OLE szerver alkalmazásaink is – ezt a szabványt követik. A COM objektumok hálózati használatát a DCOM szoftverösszetevő biztosítja.

Komponens	Jellemző	Érték
DBJegyekHelyi:TDatabase	AliasName	DBJegyek
	DatabaseName	DBJegyekHelyi
	HandleShared	True
	LoginPrompt	False *{ne jelenjen meg a bejelentkezési ablak}*
	Params	USER NAME=SYSDBA PASSWORD=masterkey
tblTantargy:TTable	DatabaseName	DBJegyekHelyi
	TableName	Tantargy

6. Jelöljük ki a táblakomponenst, majd hívjuk meg a gyorsmenüjéből az *Export tblTantargy from data module* parancsot.



Hatására a rendszer legenerált a kódban egy *Get_tblTantargy* nevű metódus. Ez az adatmodul interfészének része, ezen keresztül érik el a kliens alkalmazások a *tblTantargy* tábla tartalmát. Ez a metódus az interfészosztály állományába (JegyekSzerver_TLB.PAS) is automatikusan beíródott.

```
function TDBJegyekSzerver.Get_tblTantargy: IProvider;
begin
  Result := tblTantargy.Provider;
end;
```

7. Lépjünk át az alkalmazás űrlapjára. Helyezzünk el rajta egy *lKliensekSzama:TLabel* komponenst. Ebben fogjuk számolni az adatkiszolgálóra „felcsatlakozott" kliens alkalmazásokat. A kliensek számolását egyszerűen csak a látvány kedvéért végezzük, semmi köze sincs az adatkiszolgáló logikájához.

Komponens	Jellemző	Érték
frmJegyekSzerver: TfrmJegyekSzerver	Caption	JegyekSzerver
lKliensekSzama:TLabel	Caption	0 *{kezdetben nincs egyetlen kliens sem}*

8. A *TfrmJegyekSzerver* űrlaposztályban vezessünk be két nyilvános metódust: az *IncJegyekSzama* eggyel növelni fogja, a *DecJegyekSzama* pedig eggyel csökkenteni fogja a címke feliratát.

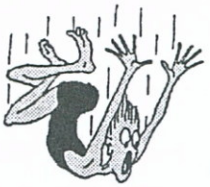
```
procedure TfrmJegyekSzerver.IncKliensekSzama;
begin
  lKliensekSzama.Caption:=
                IntToStr(StrToInt(lKliensekSzama.Caption)+1);
end;
```

```
procedure TfrmJegyekSzerver.DecKliensekSzama;
begin
  lKliensekSzama.Caption:=
                IntToStr(StrToInt(lKliensekSzama.Caption)-1);
end;
```

9. Lépjünk át az adatmodulra. Az *OnCreate* eseményben hívjuk meg az *IncJegyekSzama* űrlapmetódust, az *OnDestroy*-ban pedig a *DecJegyekSzama*-t. Ezzel fogjuk figyelni a kliensek adatmodulunkra történő csatlakozását, majd kijelentkezését.

```
procedure TDBJegyekSzerver.DBJegyekSzerverCreate(Sender: TObject);
begin
  frmJegyekSzerver.IncKliensekSzama;
end;

procedure TDBJegyekSzerver.DBJegyekSzerverDestroy(Sender: TObject);
begin
  frmJegyekSzerver.DecKliensekSzama;
end;
```

10. Futtassuk az alkalmazást! Ezzel regisztráltuk adatkiszolgálónkat (adatszerverünket). Rögtön be is zárhatjuk. A következő pontban megírjuk a kliens alkalmazást is. Az első futtatott kliens majd automatikusan el fogja indítani a szervert (adatkiszolgálót).



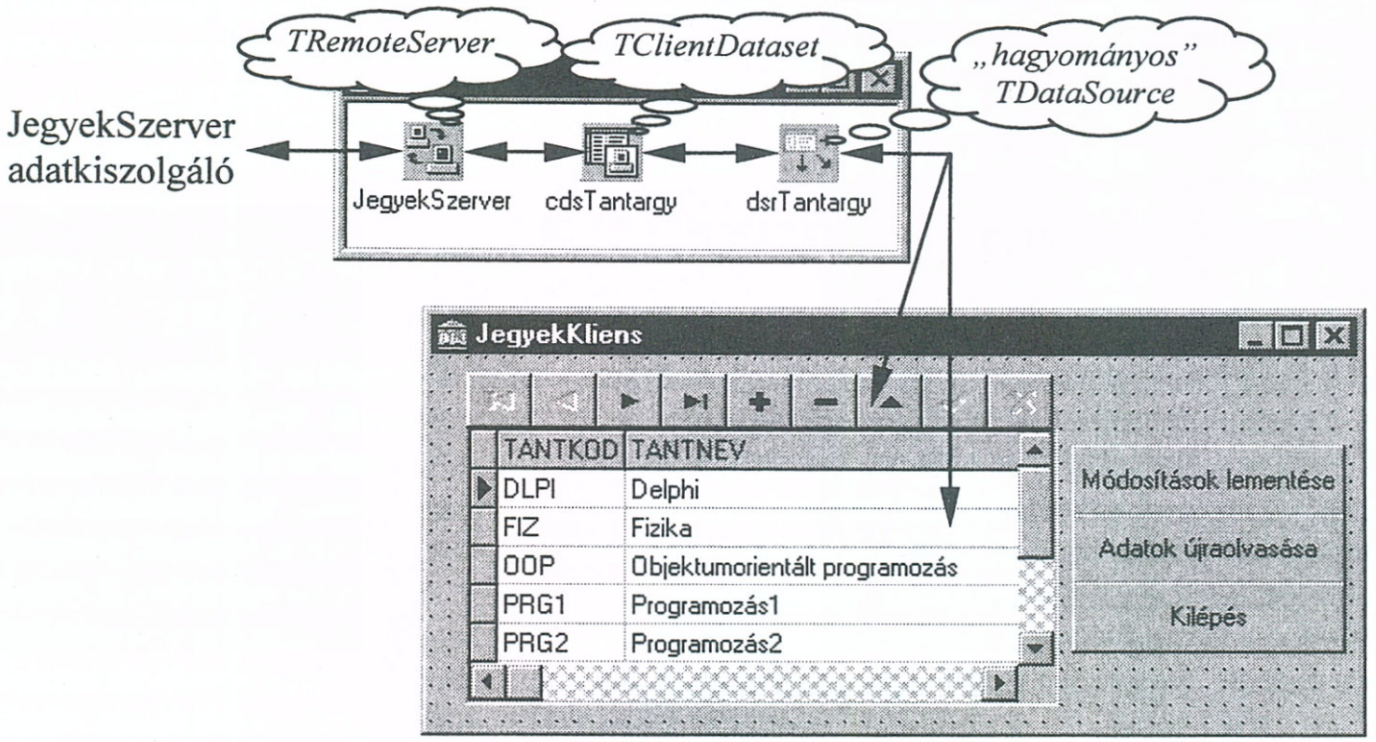
Indítsuk el a *Start* menüből a DCOMCNFG.EXE konfiguráló programot. A regisztrált COM objektumok között a mi *DBJegyekSzerver* objektumunk is jelen van. A *Properties* gombra kattintva megjelennek a kiválasztott COM objektum jellemzői. A DCOMCNFG programmal tulajdonképpen ezen objektumok hálózati elérésének módozatait állíthatjuk be.
A DCOMCNFG működéséről bővebb információk a Microsoft forrásaiban (például support.microsoft.com, TechNet...) találhatók.

Figyelem!
A Windows 95-be alapban nincs beépítve a DCOM támogatás. Ellenben ez letölthető a Microsoft honlapjáról.

19.3 A kliens alkalmazás elkészítése

A kliens alkalmazás lényegi része a felhasználói felület. A *Tantargy* tábla tartalmát egy rácsban fogjuk megjeleníteni. Mi legyen a rács adatforrása? Most adatforrásként nem *TDatabase* és *TTable* komponenseket használunk, hanem speciális komponenseket, melyek képesek a középső rétegű adatkiszolgálókkal fenntartani a kapcsolatot: adatokat fogadnak ezektől, valamint adatokat küldenek feléjük.

És még egy érdekesség: leendő kliens alkalmazásunk önálló EXE állomány lesz, vagyis nem lesz szüksége a BDE-re.



19.3. ábra. A kliens alkalmazás szerkezete

Lépések:

1. Kezdjünk egy új alkalmazást. Hozzunk benne létre egy adatmodult (egy hagyományos adatmodult: *File/New Data Module*).

2. Helyezzünk el az adatmodulon egy *TRemoteServer*, egy *TClientDataset* és egy *TDataSource* komponenst. A *TRemoteServer* kapcsolódik a távoli adatszolgáltatóhoz, a *TClientDataset* a *RemoteServer*-ből egy adott adatforrást képvisel, a *TDataSource* pedig a hagyományos módon lehetővé teszi egy adathalmaz (most a *ClientDataset*) megjelenítését. Állítsuk be jellemzőiket a következő táblázat alapján:

Komponens	**Jellemző**	**Érték**
JegyekSzerver: TRemoteServer	ServerName	JegyekSzerver.DBJegyekSzerver
	ComputerName	Az adatkiszolgáló gépének neve *{nálam ez is Pentike}*
	Connected	True *{ezzel kapcsolódunk az adatkiszolgálóra}*
cdsTantargy: TClientDataset	RemoteServer	JegyekSzerver
	ProviderName	tblTantargy
	Active	True *{az adathalmaz megnyitása}*
dsrTantargy: TDataSource	DataSet	cdsTantargy

A *JegyekSzerver:TRemoteServer* komponens *Connected* jellemzőjének igazra állításával rákapcsolódunk az adatkiszolgálóra. Ekkor alkalmazásunk elindítja az előző pontban írt középső rétegű alkalmazást. Ablakában máris látható: *Kliensek száma = 1.*

3. Lépjünk át az űrlapra. Helyezzük el rajta a rácsot és a navigátorsort, valamint három gombot (lásd 19.3. ábra).

4. A rácsot és a navigátorsort irányítsuk az adatmodulon levő *dsrTantargy* adatforrásra. A rácsban máris láthatóvá válnak a felvitt tantárgyak.

5. Kliens alkalmazásunk adatai a *DBJegyekSzerver* adatkiszolgálótól származnak. Ezek az adatok betöltődnek a kliens alkalmazás memóriájába. Tegyük fel, hogy módosítunk egy rekordot. Ez a módosítás postázáskor nem íródik be automatikusan az adatbázisba, hanem mindaddig a kliens gép memóriájában marad, amíg egy speciális metódust meg nem hívunk. Tehát lehetőség van arra, hogy az adatokon végzett módosításokat a kliens alkalmazás ne rekordonként küldje el az adatbázis-szerver felé, hanem rekordcsoportonként. Egy adathalmaz adatainak tényleges mentése nem postázáskor történik, hanem csak akkor, amikor meghívjuk az adathalmaz *ApplyUpdates* metódusát. Ezek az adatok a tényleges mentésig a gép cache-memóriájában találhatók. Ennek a technikának köszönhetően a hálózati adatforgalom jelentősen csökkenhet.

   Az űrlapon felvett két gomb az adatok tényleges lementését, illetve visszatöltését, újraolvasását célozzák meg. Ennek érdekében gépeljük be a következő kódrészletet:

```
procedure TfrmJegyekKliens.btnMentesClick(Sender: TObject);
begin
  DM.cdsTantargy.ApplyUpdates(-1);
end;

procedure TfrmJegyekKliens.btnUjraOlvasClick(Sender: TObject);
begin
  DM.cdsTantargy.Close;
  DM.cdsTantargy.Open;
end;
```

   A Kilépés gombra fejeződjön be a program.

```
procedure TfrmJegyekKliens.btnKilepesClick(Sender: TObject);
begin
  Close;
end;
```

6. Mentsük le az alkalmazást.

7. Ha most a Delphi környezetből elindítjuk kliens alkalmazásunkat, akkor azt fogjuk tapasztalni, hogy az adatkiszolgáló is megjelenik, ablakában pedig két klienst mutat: az egyik kliens a Delphi (mivel ott már tervezési időben is létrejött a kapcsolat), a másik pedig a frissen futtatott alkalmazás. Indítsuk el még néhány példányban a kliens alkalmazást. Végezzünk különböző adatmódosításokat. Ha ezeket csak postázzuk, és

nem mentjük le ténylegesen (a gombbal), akkor az adatok újraolvasásánál a módosítások elvesznek.

8. A fejlesztett kliens alkalmazás elhelyezhető egy másik gépen is. A célgépre csupán két állományt kell vinnünk: az alkalmazás EXE-jét, valamint a DBCLIENT.DLL állományt.

19.4 Végszó

Az előző pontokban felépített alkalmazásban a középső rétegnek adatkiszolgáló (***Data Broker***) szerepe volt. A lehetőségeket itt még messze nem merítettük ki. A *middle-tier* alkalmazásban az alkalmazás-logikát is implementálhatjuk, így ennek módosításakor csak egy központi helyen kell változtatnunk (nem úgy, mintha ez a kliensben lenne). Az alkalmazás-logikának a középső rétegbe való elhelyezése azt is jelenti, hogy a különböző megszorítások, ellenőrzések most már itt futnak le, nem kell a hibás adatoknak az adatbázisig eljutniuk. Ez a hálózati forgalom csökkentéséhez, valamint a hardver-erőforrások jobb kihasználásához vezet. Az alkalmazás-logikát is magába foglaló középső réteget még ***Constraint Broker***-nek is nevezzük.

Egy másik lehetőség a több rétegű alkalmazásokban az ***off-line*** üzemmód: ez azt jelenti, hogy egy kliens az adatokat akár helyben is tárolni tudja. Ennek akkor van nagy jelentősége, amikor szeretnénk ugyan egy központi adatbázis adataival dolgozni, de ez nem áll állandó jelleggel a rendelkezésünkre. Ekkor egy laptop-ban lementjük helyben az adatokat, körbe járjuk a világot, a helyi adatokon módosításokat eszközölünk, majd amikor újra „adatbázis-közelbe" kerülünk, lehetőségünk van az adatbázisba is átvezetni összegyűjtött adatainak. Persze ez csak nem-kritikus alkalmazásoknál használható, amikor az egyes adatok inkonzisztenciája nem jár végzetes következményekkel (nem „***Online Transaction Processing***"– OLTP-jellegű adatfeldolgozás). Egy egyszerű példával élve: egy banki alkalmazásban az *off-line* feldolgozás lehetővé tenné egy pénzösszeg többszörös, követhetetlen felvételét. (Ha a banki automaták az adatok másolatán dolgoznának, akkor minden automatából kiüríthetnénk számlánkat.)

Még egyszer kihangsúlyoznám, hogy úgy, ahogyan a fájl-szerver és kliens/szerver architektúrák nem elsősorban adatkezelési technikák, a több rétegű alkalmazások sem kizárólagosan az adatfeldolgozásban használatosak. A hálózati alkalmazások között több ilyen jellegűt találhatunk: levelező rendszerek, csoport-kezelők, proxy-szerverek. A legismertebb talán az adatokat szolgáltató web-szerverek esete: az adatbázis egy gépen, a web-szerver egy másikon, a kliensek pedig szerte a világban találhatók.

20. Több szálon futó alkalmazások

Az első fejezetben már említettük, hogy a Windows 32 bites változataiban több szálú alkalmazásokat is készíthetünk. Azt is láthattuk, hogy a „*preemptive multitasking*" révén a rendszer a párhuzamosan futó alkalmazások párhuzamosan futó szálai között osztja ki a processzoridőt. Ugyanakkor minden szál külön üzenetsorral is rendelkezik, itt várakoznak a neki szánt üzenetek egészen feldolgozásukig.
Ebben a fejezetben megismerkedünk a több szálú alkalmazások készítésének módjával. Előbb tisztázzuk az alapfogalmakat, majd készítünk Delphiben egy adatbázist kezelő több szálú programot.

20.1 A szál (*thread*) fogalma

Az eddigi alkalmazásaink mind egy szálon futottak. Ezt az alapértelmezés szerint létrejött szálat az alkalmazás **fő szálának** (*main thread*) nevezzük. Ezen kívül alkalmazásunkban további szálakat indíthatunk, ezek kódja párhuzamosan fog lefutni. **A szál az alkalmazáson belül egy önálló műveletsorozatot képvisel.** A szál az őt elindító alkalmazás címterületét, kódját, globális adatait és jogosultságait használja, de saját veremmel és üzenetsorral rendelkezik.
Amint az 1. fejezetben láttuk, a 32 bites rendszerekben a *multitasking* nem a párhuzamosan futó alkalmazások, hanem a párhuzamos szálak között értendő. A processzor nem az alkalmazások között osztja ki a processzoridőt, hanem szálanként teszi mindezt. A Windows NT-ben pedig, ahol akár 2, 4 processzort is elhelyezhetünk gépünkben, a párhuzamos szálakat akár külön processzor is működtetheti. Első megközelítésben azt mondhatjuk, hogy a két szálon futó alkalmazás kétszer annyi idővel rendelkezik, mint egy egy szálú program. A megnövekedett időt párhuzamos feldolgozásokra használhatjuk ki. Ha egy alkalmazásban egyértelműen szétválaszthatók a párhuzamosan elvégzendő műveletek, akkor időt nyerünk azzal, hogy ezeket két vagy több külön szálon valósítjuk meg. Természetesen a szálak között szinkronizálási lehetőségek is vannak, ezek segítségével a párhuzamos szálak jelzéseket küldhetnek egymásnak, akár be is várhatják egymást. Ugyanakkor beállítható minden egyes szál prioritása is.
A több szálú programok másik alkalmazási területét a valós idejű feldolgozások jelentik. Ha például alkalmazásunknak állandóan figyelnie kell egy adott perifériát, mondjuk egy jelre várva, akkor ezt elvileg egy időzítő segítségével az alkalmazás fő szálában is megtehetjük. Ez azonban feleslegesen elrabolná az időt az alkalmazás egyéb tevékenységei elől. A megoldás egy külön, figyelő szál elindítása.

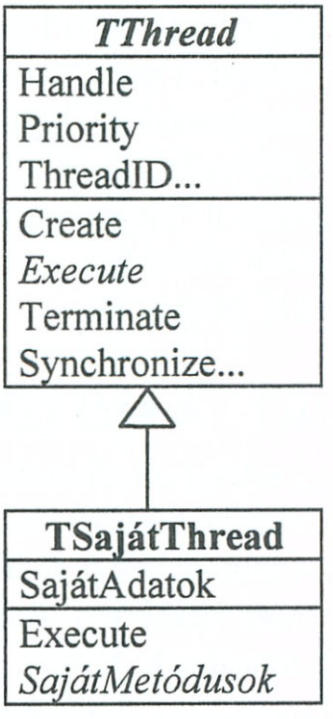
Ne gondoljuk viszont azt sem, hogy minél több a szál, annál jobb, annál gyorsabb az alkalmazás. A sok szál a processzort is nagyon leterheli. A gyakorlat pedig azt mutatja, hogy Murphynek a több szálú alkalmazásoknál még inkább igaza van. Itt olyan dolgok is komoly hibákat okozhatnak, amelyek az eddigiekben simán lefutottak. Csak indokolt esetben írjunk több szálú alkalmazást, és a szálak számát akkor se vigyük túlzásba.

20.2 Több szálú alkalmazások a Delphiben

Delphiben az újabb szálak létrehozásának alapját a *TThread* osztály képezi. Ebből kell származtatnunk a konkrét szálak osztályait. Egy utód osztály alapján akárhány objektumot létrehozhatunk, azaz egy szálosztály mintájára több, azonos feladatot ellátó szálat is indíthatunk.

Az alapelv tehát a következő:

- Kijelöljük pontosan az új szál feladatát.
- Létrehozunk egy új osztályt a *TThread* mintájára. Ebben felveszünk új adatmezőket, új metódusokat. Az új adatok inicializálásáért legtöbbször felül kell írnunk a *TThread*-ből örökölt konstruktort, a *Create*-t. Saját szálaink a *TSajátThread* osztály objektumai lesznek. A feladatukat az absztrakt *Execute* metódus felülírásával kell megadnunk. A szál, vagy rögtön a létrehozása után, vagy egy későbbi pillanatban (ez a *Create* paraméterének értékétől függ) elindítja az *Execute* metódusába írt kódot; amikor ez befejeződik, akkor a szálnak is vége.
- Delphiben van egy megkötés: a VCL (vizuális komponensek könyvtára) elemeinek metódusait és jellemzőit csak az alkalmazás fő szálában (*main thread*) szabad meghívni, átírni. Ez azt jelenti, hogyha például egy saját szálban meg akarjuk változtatni egy címke feliratát, akkor ezt a beépített *Synchronize* metódus segítségével tehetjük csak meg.



20.1. ábra.

```
{létre kell hoznunk egy metódust a címke átírására}
procedure TSajatThread.UpdateCaption;
begin
  Form1.Caption := 'Új felirat';
end;

{a konkrét átírás pedig így valósul meg:}
Synchronize(UpdateCaption);
```

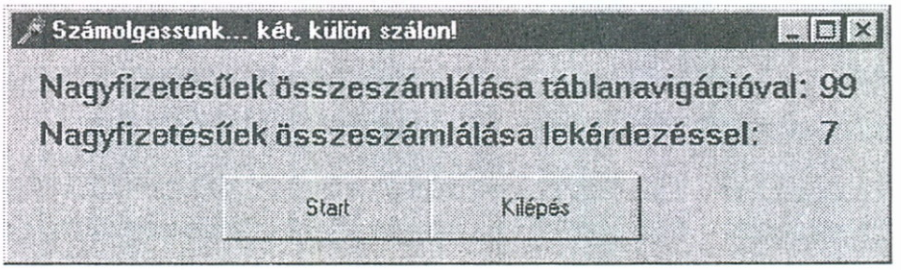
Mindezek jobban is le fognak tisztulni a következő feladat segítségével:

20.3 Több szálú adatbázisos feladat

Készítsünk egy alkalmazást, mely a táblák feldolgozásának módjait hasonlítja össze. Adott egy 10000 rekordos tábla, az *DBDEMOS:MEmploy*. Benne két mező van: *EmpNo* (Azonosító) és *Salary* (Fizetés). Számoljuk össze a 200000 Ft-nál nagyobb fizetésűeket két módszerrel: az egyikben a tábla Delphiből történő végigolvasásával, a másodikban pedig egy lekérdezéssel számoljuk össze a „gazdagokat". (Emlékezzünk vissza, ezzel a feladattal már a 8. fejezetben is találkoztunk. Ott a számolásokat egymás után indíthattuk csak el.)

A számolásokat most két külön szálon indítsuk el, annak érdekében, hogy a két módszer hatékonyságát minél jobban összevethessük.

A műveletek idejét folyamatosan jelezzük ki két címke segítségével. A számolások végeztével összevethetjük az időket.



20.2. ábra. Alkalmazásunk űrlapja

Megoldás (🖫 20_THREADS\PARADOX\PTHREADS.DPR)

A megoldáshoz a 8_NAVIG\PNAVIG.DPR alkalmazás által a DBDEMOS álnév alatt létrehozott 10000 rekordos MEMPLOY.DB táblát használjuk. Azért választunk ekkora táblát, hogy a módszerek közötti különbség nyilvánvalóbb legyen. A „nagy tábla" létrehozásának kódját biztonságból a jelen feladatba is beépítettük: a PTHREADS első indításánál megvizsgáljuk, hogy létezik-e a teszttábla. Ha még nem, akkor most hozzuk létre. A megvalósítási részletekre most nem térünk ki, mivel ezeket a 8. fejezet 6. pontjában már megismerhettük.

A megoldás három lépésből áll:

- Először megtervezzük az alkalmazás adatmodulját. Ebben elhelyezünk egy tábla és egy lekérdezés komponenst. Egyik a „navigációs" szálhoz, a másik pedig a lekérdezéseshez szükséges.
- Utána megtervezzük a szálak osztályait.
- Végül megtervezzük az alkalmazás űrlapját. Itt egy *TTimer* segítségével fogjuk mérni a szálak időtartamát.

20.3.1 Az adatmodul megtervezése

Hozzunk létre egy új alkalmazást, benne egy új adatmodult. Helyezzünk el rajta egy *tblEmployee:TTable* és egy *qEmployee:TQuery* komponenst. Mindkettő *DatabaseName* jellemzőjét állítsuk *DBDEMOS*-ra (ott van, vagy ott lesz az MEMPLOY.DB táblázat). A

táblakomponenst irányítsuk az MEMPLOY táblára (ha ez még nem létezik, akkor gépeljük be a nevét). A lekérdezés *SQL* jellemzőjébe a következők kerülnek:

```
SELECT COUNT(*) FROM MEmploy
WHERE Salary > 200000
```

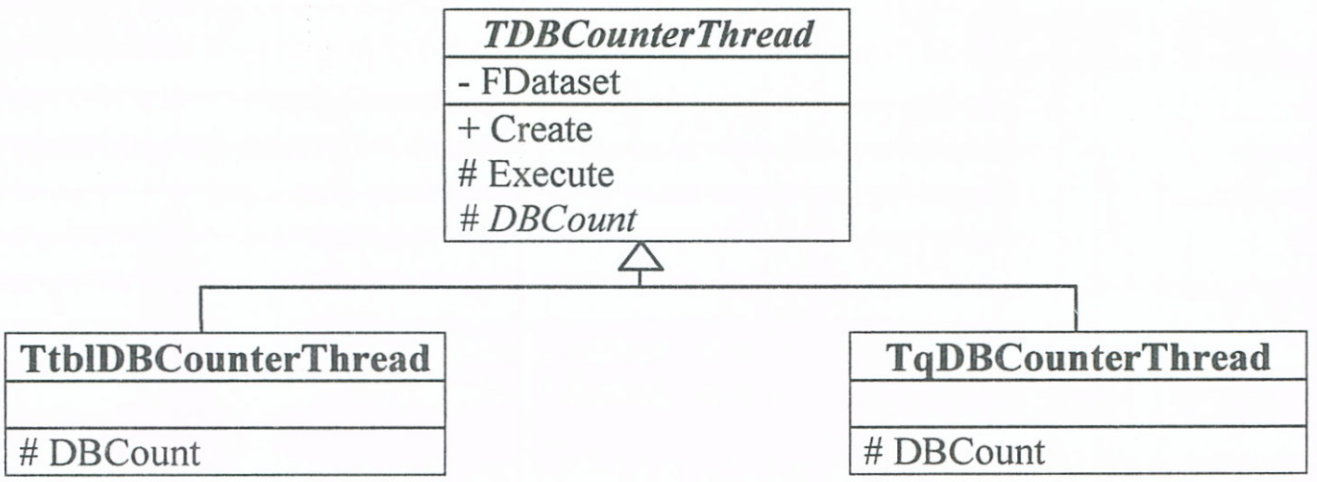
Végül mentsük le az adatmodult ⇨UDM.PAS.

20.3.2 A szálak megtervezése

Mivel két különböző összeszámolási módszerről van szó, két külön szálosztályra van szükségünk. Mindkét szálban azonban közös az a tény, hogy egy adathalmazt dolgoznak fel. Az igazi az lenne, ha a szálak létrehozásánál adnánk át paraméterként a feldolgozandó adathalmazt: táblát vagy lekérdezést. Valahogy így:

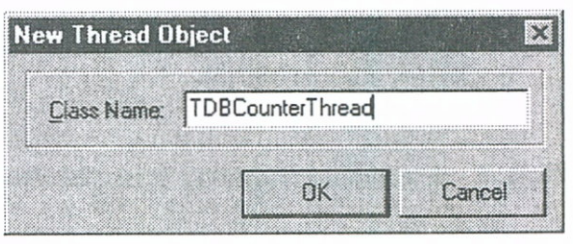
```
Thread1:= TSajatThreadOsztaly1.Create(tblEmployee)
Thread2:= TSajatThreadOsztaly2.Create(qEmployee)
```

Hozzunk létre a közös elemek számára egy *TDBCounterThread* szálosztályt (20.3. ábra). Ebben definiálhatjuk a konstruktort, és az adathalmazt tároló adatmezőt. Mindkét szál elindítja a számolást az *Execute* metódusában. A számolás számára vezessük be a *DBCount* metódust. Ez a *TDBCounterThread* osztályban még absztrakt és virtuális, az utódosztályokban azonban már tartalmazni fogja a megfelelő számolási módszer kódját.



20.3. ábra. A számolásokat végző szálak osztályhierarchia-diagramja

Hozzuk létre az új szálakat. Hívjuk meg a *File/New...* menüpontot. Válasszuk ki a *Thread Object* opciót. A megjelenő ablakba a közös szálosztály nevét írjuk.



20.4. ábra. Új szál létrehozása

A rendszer legenerálja az új szálosztály vázát. Töltsük fel tartalommal a következők alapján:

```
unit UDBThread;

interface
uses Classes, DBTables;

type
  TDBCounterThread = class(TThread)
  private
    FDataset:TDBDataset;
  protected
    procedure Execute; override;
    procedure DBCount;virtual;abstract;
  public
    constructor Create(iDataset:TDBDataSet); override;
  end;

  TtblDBCounterThread = Class (TDBCounterThread)
    protected
      Procedure DBCount;Override;
  End;

  TqDBCounterThread = Class (TDBCounterThread)
    protected
      Procedure DBCount;Override;
  End;

implementation

{****************TDBCounterThread*****************}
constructor TDBCounterThread.Create(iDataset:TDBDataSet);
begin
  inherited Create(False);
  {False ⇨ a Create után automatikusan meghívódik az Execute metódus}
  FDataset:=iDataset;
end;

procedure TDBCounterThread.Execute;
begin
  DBCount;
end;

{****************TtblDBCounterThread*****************}
Procedure TtblDBCounterThread.DBCount;
Var Szamlalo:Integer;
begin
  Try
    FDataset.Open;
```

```
    With FDataset Do
      begin
          First;
          Szamlalo:=0;
          While Not EOF Do
            begin
              if FieldByName('Salary').Value>200000 Then
                Inc(Szamlalo);
              Next;
            end;
      end;
  Finally
    FDataset.Close;
  End;
end;

{****************TqDBCounterThread*****************}
Procedure TqDBCounterThread.DBCount;
begin
  Try
    FDataset.Open;
  Finally
    FDataset.Close;
  End;
end;

end.
```

20.3.3 Az űrlap megtervezése

Helyezzük el az űrlapon az időkijelző címkéket: *tblLabel* és *qLabel.* A *btnStart* gombbal elindítjuk a szálakat, a *btnKilepes*-re pedig kilépünk a programból.
Ugyancsak az űrlap felelőssége a két elindított szál időtartamának figyelése, címkéik frissítése. Ezt a feladatot egy *Timer:TTimer* segítségével fogja ellátni. Az időzítő századmásodpercenként lekérdezi a szálak állapotát. Ha a szál még mindig fut, akkor annak címkéjét eggyel megnöveli. A szálakat a *btnStart.OnClick*-ben hozzuk létre, állapotukat pedig a *Timer.OnTimer*-ben kérdezzük le. Ahhoz, hogy a szálak mindkét metódusban elérhetők legyenek, vegyünk fel az űrlaposztályban két szálobjektumot: *Thread1* és *Thread2.*
A *Start* és a *Kilépés* feliratú gombokat a szálak futásának idejére le kell tiltanunk (ne lehessen se új szálakat indítani, se kilépni). Ennek érdekében bevezetjük a *ThreadsRunning* változót, melyben a futó szálak számát tároljuk. Indításkor beállítjuk kettőre. Amikor valamelyik szálnak vége (*OnTerminate*), meghívódik a *ThreadDone* metódus. Ebben csökkentjük a futó szálak számát. Amikor elértük a nullát, akkor mindkét szálnak vége, tehát úgy a *btnStart*, mind a *btnKilepes* gombot újból engedélyezzük.

Nézzük a kódot:

```
unit Ufrm;

interface
uses  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
      Dialogs, ExtCtrls, StdCtrls, uDBThread, Grids, DBGrids;

type
  TfrmThreads = class(TForm)
    ...
  private
    Thread1, Thread2: TDBCounterThread;
    ThreadsRunning:Integer;
    procedure StopperBe;
    procedure ThreadDone(Sender:TObject);
  end;

var
  frmThreads: TfrmThreads;

implementation
uses UDM;
{$R *.DFM}

procedure TfrmThreads.StopperBe;
begin
  tblLabel.Caption:='0';
  qLabel.Caption:='0';
  Timer.Enabled:=True;
end;

procedure TfrmThreads.ThreadDone(Sender:TObject);
begin
  Dec(ThreadsRunning);
  MessageBeep(1);
  If ThreadsRunning = 0 Then
    begin
      btnStart.Enabled:=True;
      btnKilepes.Enabled:= True;
    end
end;

procedure TfrmThreads.btnKilepesClick(Sender: TObject);
begin
  Close;
end;
```

```
procedure TfrmThreads.btnStartClick(Sender: TObject);
begin
  StopperBe;
  Thread1:= TtblDBCounterThread.Create(DM.tblEmployee);
  Thread2:= TqDBCounterThread.Create(DM.qEmployee);
  Thread1.OnTerminate:= ThreadDone;
  Thread2.OnTerminate:= ThreadDone;
  btnStart.Enabled:=False;
  btnKilepes.Enabled:=False;
  ThreadsRunning:=2;
end;

procedure TfrmThreads.TimerTimer(Sender: TObject);
var exitcode:Integer;
begin
  GetExitCodeThread(thread1.handle,exitcode);
  if exitcode= STILL_ACTIVE Then
    tblLabel.Caption:= IntToStr(StrToInt(tblLabel.Caption)+1);
  GetExitCodeThread(thread2.handle,exitcode);
  if exitcode= STILL_ACTIVE Then
    qLabel.Caption:= IntToStr(StrToInt(qLabel.Caption)+1);
end;

end.
```

Egy szál állapotát a *GetExitCodeThread* API függvénnyel kérdezzük le. Első paramétereként átadjuk a szál „fogantyúját" (*handle*), a másodikban pedig visszakapjuk az állapotát. Ha a szál még mindig fut, akkor ennek értéke STILL_ACTIVE (=259) lesz.

Tesztelje le az alkalmazást. Első alkalommal még elég lassú a számolás, a továbbiakban viszont felgyorsul. Ez a „cache"-elés eredménye. A két módszer közötti különbség egyértelmű.

A Windows NT-t használó Olvasóimnak ajánlom, hogy a *Task Manager (Feladatkezelő) Processes (Folyamatok)* oldalán kövessék nyomon a szálak számának alakulását az alkalmazásunk futásának ideje alatt. Figyelem, a *Task Manager*-ben alapértelmezés szerint nincs feltüntetve a folyamatokon belüli szálak száma. Ezt a *View/Select Columns...* (*Nézet/Oszlopok kiválasztása*) párbeszédablakban a *Thread Count* (*Szálak*) jelölőnégyzet kipipálásával jeleníthetjük meg.

20.3.4 A feladat Interbase-es megvalósítása

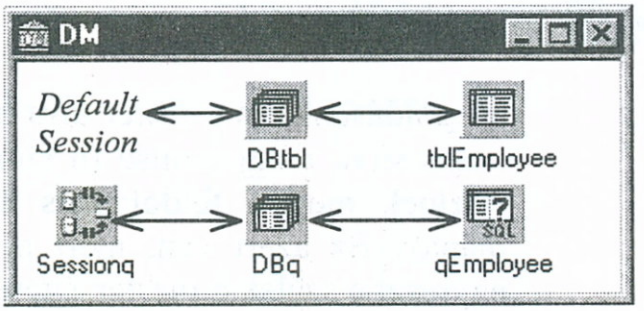
Alakítsuk át a feladatot úgy, hogy az ne Paradox, hanem Interbase táblában számolja össze a „gazdagokat".

Megoldás (🖫 20_THREADS\INTERBASE\PTHREADS.DPR)

Nem kell túl sokat változtatnunk, figyelembe véve azt a tényt, hogy az alkalmazásba be van építve a nagy tábla létrehozásának kódja. Ebben egyszerűen át kell írnunk a kódban az új tábla álnevét: *DatabaseName := IBLOCAL*. Amikor először elindítjuk, az alkalmazás létre fogja hozni teszttáblánkat az *IBLOCAL* adatbázisban.

Figyelem! A tábla létrehozása hosszadalmas folyamat, 1-2 percig is eltarthat.

Az adatmodulon át kell írnunk a *tblEmployee* és *qEmployee* komponensek *DatabaseName* jellemzőjét. Azonban ennyi nem elég. Delphiben, ha egy adatbázis azonos fizikai tábláját le szeretnénk kérdezni két párhuzamos szálon, akkor a szálaknak külön *Session* komponensekkel kell rendelkezniük (lásd 8. fejezet 2. pont). Minden adatbázisos alkalmazásban alapértelmezés szerint létrejön egy *Session,* ez a *Default Session*. Az első szál ezt használja. A második szálnak viszont nekünk kell egy *TSession* komponenst létrehoznunk. És mivel Interbase adatbázisról van szó, szükség van még két *TDatabase* komponensre is. Helyezzük el ezeket az adatmodulon, majd állítsuk be jellemzőiket a táblázatnak megfelelően:



20.5. ábra. Az új adatmodul

Komponens	Jellemző	Érték
DBtbl:TDatabase	AliasName	IBLOCAL
	DatabaseName	DBtbl
	LoginPrompt	False *{ne jelenjen meg a bejelentkezési ablak}*
	Params	USER NAME=SYSDBA PASSWORD=masterkey
	Connected	True
tblEmployee:TTable	DatabaseName	DBtbl
	TableName	MEmploy
Sessionq:TSession	SessionName	Sessionq
	Active	True

Komponens	Jellemző	Érték
DBq:TDatabase	SessionName	Sessionq
	AliasName	IBLOCAL
	DatabaseName	DBq
	LoginPrompt	False *{ne jelenjen meg a bejelentkezési ablak}*
	Params	USER NAME=SYSDBA PASSWORD=masterkey
	Connected	True
qEmployee:TQuery	DatabaseName	DBq
	SesionName	Sessionq

Futassuk le az alkalmazást. Az arányok megmaradtak, de a számlálás több időbe telik.

Valami azért nem teljesen jó ebben a feladatban. Egy logikai bukfenc található benne. Mi az?

Megoldásunkban a *Timer* arra megfelel, hogy a két módszer közötti különbséget számosítva megjelenítse (a címkék formájában), azonban ezek az értékek nem felelnek meg a feldolgozás tényleges századmásodpercekben mért időtartamának. Ez azért van, mert a címkék frissítése eleve időigényes feladat, így egyrészt a szálakat mesterségesen visszatartjuk, lassítjuk, másrészt pedig a *timer*-esemény lefuttatásának számosságában sem lehetünk bizonyosak.

A problémának több megoldása is van. Vagy lekérdezzük az időpontot a szálak indításakor és befejezésekor, vagy pedig bevezetünk egy újabb, időfigyelő szálat. Az első megoldásnak az a hátránya, hogy így nem tudnánk a feldolgozás közben jelezni az időt. A második megoldás viszont kicsit bonyolultabb. Az új, időfigyelő szálban az lenne a jó, hogy nem „szívná el a levegőt" a dolgozó szálaink elől, így valószínű reális eredményt mutatna.

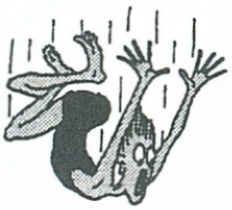
E két megoldást Önökre bízom. Jó munkát!

Búcsúzóul jó munkát és sok sikerélményt kívánnak a Delphi használata során:

Azért nem is voltak olyan keservesek ezek a feladatok...

Megoldás (🖫 De azért jó, ha megvan rájuk a megoldás...)

Mi? Hogy? Nem egészen értem, de érdekesnek tűnik!



Már nem félek a mély víztől... megtanultam úszni.

Aztán csak így tovább!

Irodalomjegyzék

[1] Angster Erzsébet: Programozás tankönyv I.
Magánkiadás, 1995

[2] Angster Erzsébet: Programozás tankönyv II.
Magánkiadás, 1995

[3] Angster Erzsébet: Az objektumorientált tervezés és programozás alapjai
Magánkiadás, 1997

[4] Kupcsikné Fitus Ilona: Adatbázisok példatár
LSI, 1997

[5] Szelezsán János: Adatbázisok
LSI, 1997

[6] Gyenes László, Juhos Margit: Az SQL alapjai
aLapok Könyv és Lapkiadó, 1992

[7] Juhász Mihály, Kiss Zoltán, Kuzmina Jekatyerina, Sölétormos Gábor, Dr. Tamás Péter, Tóth Bertalan: Delphi – Út a jövőbe
ComputerBooks, 1996

[8] Kertész László: Delphi – Környezet és nyelv
Magánkiadás, 1995

[9] Gary Cornell: Delphi – Tippek és trükkök
Panem-McGraw-Hill, 1997

[10] Vámossy Zoltán: Delphi a gyakorlatban – Mintafeladatok megoldással
Szak Kiadó, 1997

[11] Henderson, Ken: Client/Server Developer's Guide
SAMS Publishing, 1997

[12] Teixeira, Steve - Pacheco, Xavier: Delphi Developer's Guide
SAMS Publishing, 1996

[13] Orfali, Robert - Harkey, Dan - Edwards, Jeri: The Essential Client/Server Survival Guide, Wiley Computer Publishing, 1996

[14] Borland Delphi for Windows 95 & Windows NT: User's Guide
Borland International, 1995

[15] Borland Delphi for Windows 95 & Windows NT: Component Writer's Guide
Borland International, 1995

[16] Borland Delphi for Windows 95 & Windows NT: Database Application Developer's Guide, Borland International, 1995

[17] Borland Interbase: Data Definition Guide
Borland International, 1992

Tárgymutató

A

Ablakfüggvény
  Delphiben 26
  Windowsban 13
Abort metódus
  TPrinter 149
Absztrakt metódus 38
Access/Assign metódusok 44
Active jellemző
  TTable 216
ActiveControl jellemző
  TScreen 94
ActiveMDIChild jellemző
  TForm 110
ActivePage jellemző
  TNoteBook 101
  TPageControl 100
  TTabbedNoteBook 101
Adatelérési komponensek 168
  Áttekintés 175
Adathalmazok
  Állapotai 182
  Mozgás 184
  Nyitás, zárás 183
Adathozzáférés korlátozása 41
Adatmegjelenítési komponensek 168, 225
  Használata 225
Adatmodul
  Delphi 1 170
  Delphi 2,3 171
  Használata 171
Adatszótár (Data Dictionary) 321
Add metódus
  TList 143
  TMenuItem 77
  TStrings 81
AfterOpen esemény
  TDBDataset 199
AfterPost esemény
  TDBDataset 199
AliasName jellemző
  TDatabase 180
Align jellemző 62
Alignment jellemző
  TEdit 66
  TField 211
Alkalmazás-logika 157, 320
Állományműveletek 54
AllowAllUp jellemző 70
AllowGrayed jellemző 71
Álnév (Alias) 165
AnsiString típus 57
API 6, 7
Append metódus
  TDBDataset 190
Application objektum 21
ArrangeIcons metódus
  TForm 112
As 46
Assign metódus
  TClipboard 379
AsString, AsInteger... jellemzők
  TField 211
Auto-create űrlapok 105
AutoEdit jellemző
  TDataSource 221
AutoPopup jellemző 76

B

BDE (Borland Database Engine) 163
BDE Administrator 167
BeforeDelete esemény
  TDBDataset 189, 199
BeforeOpen esemény
  TDBDataset 199
BeforePost esemény
  TDBDataset 199
BeginDoc metódus

TPrinter 149
BeginDrag metódus 91
BOF jellemző 184
BorderIcons jellemző
TForm 109
BorderStyle jellemző
TForm 110
Business logic 320

C

C karakterlánc 56
Calculated field 205, 207
Cancel jellemző 68
Cancel metódus
TDBDataset 188
Capacity jellemző
TList 143
Caption jellemző
TButton, TBitBtn 68
TForm 109
TMenuItem 76
Cascade metódus
TForm 111
Checked jellemző
TCheckBox 71
TMenuItem 77
Class típus 33
Clear metódus
TStrings 81
ClearSelection metódus 67
ClientHeight jellemző 86
ClientWidth jellemző 86
Clipboard objektum 379
Close metódus
TDatabase 181
TForm 111
Color jellemző 63
COM 394
Commit metódus
TDatabase 181
Connected jelemző
TDatabase 180
Constraint Broker 399
ConstraintErrorMessage jellemző
TField 212
Constructor 34
CopyToClipboard metódus
TEdit 67
Count jellemző
TList 143
TMenuItem 77
TStrings 80
Create 34, 36
CREATE DOMAIN 310
CREATE GENERATOR 312
CREATE INDEX 312
CREATE PROCEDURE 317
CREATE TABLE 311
CREATE TRIGGER 314
CREATE VIEW 319
Cursor jellemző
TScreen 94
Cursors jellemző
TScreen 94
CustomConstraint jellemző
TField 212
CutToClipboard metódus
TEdit 67

D

Data Access *lásd* adatelérési komponensek
Data Broker 391
Data Controls *lásd* adatmegjelenítési komponensek
Data Dictionary 321
DataBase Desktop 168
Database Explorer 167
DatabaseName jellemző
TDatabase 180
TQuery 270
TTable 216
DataSet jellemző
TDataSource 221
DataSource jellemző
TQuery 270
Date függvény 32
DateTimeToStr függvény 33

DateToStr függvény 33
DCOM 394
DDE 4, 380
  NetDDE 385
DecodeDate eljárás 33
DecodeTime eljárás 33
Default jellemző 68
Delete metódus
  TDBDataset 188
  TList 144
  TStrings 81
Destroy 36
Destruktor 36
DFM, űrlapterv állomány 17
  Szerkezete 22
Dialogs 107
Dinamikus adatcsere lehetőségek 4
Dinamikus metódusok 37
DisplayFormat jellemző
  TField 211
DisplayLabel jellemző
  TField 211
DisplayWidth jellemző
  TField 212
DLL 5, 18
DMT 37
Domains 310
DOS-Windows kapcsolat 6
Down jellemző
  TSpeedButton 70
DPR, projektállomány 17
  Szerkezete 20
Drag&Drop technika 91
DragCursor jellemző 63, 91
DragMode jellemző 91
Draw metódus
  TCanvas 132

E

EAbort kivétel 53
EConvertError kivétel 33, 51
Edit metódus
  TDBDataset 187
EditFormat jellemző
  TField 211
EditMask jellemző
  TField 212
Egy rétegű architektúra 160
EInOutError kivétel 54
EIntOverflow kivétel 54
EInvalidTypeCast kivétel 47
Ellipse metódus
  TCanvas 132
Enabled jellemző 63
  TMenuItem 77
EndDoc metódus
  TPrinter 149
EOF jellemző 184
ERangeError kivétel 54
Esemény 9
Eseménykezelő metódus 40
Exception handling 48
ExecSQL metódus
  TQuery 271
EXECUTE PROCEDURE 318
EZeroDivide kivétel 51

F

Fájl-szerver (File server) architektúra 159
FieldName jellemző
  TField 210
Fillrect metódus
  TCanvas 131
Filter jellemző
  TDBDataset 194
Filtered jellemző
  TDBDataset 195
FilterOptions jellemző
  TDBDataset 195
FindKey metódus
  TTable 193
FindNearest metódus
  TTable 194
First metódus 184
FloatToStr függvény 33
Font jellemző 63
FormStyle jellemző
  TForm 110

Fő-segéd űrlapok 222
Free 36
Friend 42
Futás alatti kötés 37

G

GENERATOR 312
Glyph jellemző 69
GRANT 319
GroupIndex jellemző
  TMenuItem 77
  TSpeedButton 70
GUI (Graphical User Interface) 6

H

HasFormat metódus
  TClipboard 379
Height jellemző 63
  TScreen 95

I

IDAPI32.CFG 165
IndexFieldNames jellemző
  TTable 216
IndexName jellemző
  TTable 216
Inherited 40
InputBox függvény 106
Insert metódus
  TDBDataset 190
  TList 144
  TMenuItem 77
  TStrings 81
InstallShield Express 368
Interbase Windows ISQL 309
IntToStr függvény 33
Is 46
IsNull jellemző
  TField 210
ItemAtPos metódus
  TListBox 73
ItemIndex jellemző
  TComboBox 74
  TListBox 72
  TRadioGroup 72
Items jellemző
  TComboBox 74
  TList 143
  TListBox 72
  TMainMenu 75
  TMenuItem 77
  TPopupMenu 76
  TRadioGroup 71

J

Jelentések 291, 293
Jellemző 42
Jogosultságok (adatbázisban) 319

K

Két rétegű architektúra 162
Kind jellemző 69
Kivételek kezelése 48
Kliens/szerver (Client/Server) architektúra 160
Kliens/szerver adatbázis-kezelés 307
Komponenscsomagok 336
Komponensek fejlesztése 331
Konstruktor 34
Konverziós függvények (szám – karakterlánc – dátum) 33

L

Last metódus 184
Left jellemző 63
Lines jellemző
  TMemo, TRichEdit 67
LineTo metódus
  TCanvas 131
LoadFromFile metódus

TPicture 128
TStrings 81
Local Interbase Server 307
Locate metódus
TDBDataset 190
LoginPrompt jellemző
TDatabase 180
Lookup field 205, 206
Lookup metódus
TDBDataset 191

M

Master-detail *lásd* fő-segép űrlapok
MasterFields jellemző
TTable 216
MasterSource jellemző
TTable 216
MaxLength jellemző 66
MDI alkalmazás 108
készítése 116
MDIChildCount jellemző
TForm 110
MDIChildren jellemző
TForm 110
Menük használata 74
Message 40
Message (kivételobjektum jellemzője) 53
Message (Windows üzenet) 9
Message loop *lásd* üzenetkezelő ciklus
MessageDlg függvény 106
Metódusok 36
Microsoft Help Workshop 355
Modális űrlapok 105
ModalResult jellemző
TButton, TBitBtn 68
TForm 110
Mode jellemző
TPen 130
MoveTo metódus
TCanvas 131
MultiSelect jellemző
TListBox 72
Multitasking 15
Nonpreemptiv 15
Preemptiv 16
Multi-tier 162, 391

N

Name jellemző 60
Natív meghajtó 163
NetDDE (hálózati DDE kommunikáció) 385
NewPage metódus
TPrinter 149
Next metódus 184
TForm 111
Nézetek 319
Nyilvános 41
Nyomtatás 148
Grafikus 148
Karakteres 149

O

Object Pascal
Áttekintés 31
Karakterláncok 56
Osztálymodell 33
Object típus 33
Objects jellemző 139
TStrings 80
ODBC 164
DSN (Datasource Name) 166
Meghajtó 163
OLE 4, 387
OLTP 399
OnActivate esemény
TForm 112
OnCalcFields esemény
TDBDataset 199
OnChange esemény
TEdit 67
TPageControl 100
TTabbedNoteBook 101
TTabControl 100
TTabset 102
OnChanging esemény

TPageControl 100
TTabControl 100
OnClick esemény 64
TTabbedNoteBook 101
TTabset 102
OnClose esemény
TForm 112
OnCloseQuery esemény
TForm 112
OnCreate esemény
TForm 112
OnDblClick esemény 64
OnDeactivate esemény
TForm 112
OnDeleteError esemény
TDBDataset 200
OnDestroy esemény
TForm 112
OnDragDrop esemény 64
OnDragOver esemény 64
OnDrawItem esemény
TListBox 139
OnDropDown esemény
TComboBox 74
OnEditError esemény
TDBDataset 200
OnEndDrag esemény 64
OnEnter esemény 65
OnExit esemény 65
OnFilterRecord esemény
TDBDataset 194
OnKeyDown esemény 65
OnKeyPress esemény 65
OnKeyUp esemény 65
OnLogin esemény
TDatabase 181
OnMouseDown esemény 64
OnMouseMove esemény 64
OnMouseUp esemény 64
OnNewRecord esemény
TDBDataset 199
OnPageChanged esemény
TNoteBook 101
OnPaint esemény
TForm 112
OnPopup esemény 76
OnPostError esemény
TDBDataset 200
OnShow esemény
TForm 112
OnStartDrag esemény 64
OnStateChange jellemző
TDataSource 221
OnValidate esemény
TField 212
Open metódus
TDatabase 181
TQuery 271
Osztályoperátorok 46
Override 37
Owner jellemző 60

P

Packages 336
PageIndex jellemző
TNoteBook 101
TTabbedNoteBook 101
Pages jellemző
TNoteBook 101
TTabbedNoteBook 101
ParamByName jellemző
TQuery 280
Paraméteres lekérdezések 279
Params jellemző
TDatabase 180
TQuery 270, 279
Párbeszédablakok 107
Parent jellemző 61
PAS, az űrlaphoz tartozó egység
Szerkezete 24
PAS, az űrlaphoz tartozó egység 17
PAS, rutinkönyvtárak 18
PASSTHRU SQL 269
PasteFromClipboard metódus
TEdit 67
PChar típus 56
Picture jellemző
TImage 128
Polygon metódus
TCanvas 132

Polyline metódus
TCanvas 131
PopupMenu jellemző 63
Position jellemző
TForm 110
Post metódus
TDBDataset 187
Prepare metódus
TQuery 271
Print metódus 111
Prior metódus 184
Privát 41
Private 41
ProcessMessages 15
Property 44, 335
Protected 41
Public 41
Publikált 41
Published 41

Q

QuickReport komponenscsalád 293

R

Raise utasítás 54
Read/Write metódusok 44
ReadOnly jellemző
TField 212
TTable 216
Rectangle metódus
TCanvas 131
Remove metódus
TMenuItem 77
Reports 291
RequestLive jellemző
TQuery 270
Result (függvény eredménye) 48
REVOKE 319
Rollback metódus
TDatabase 181
RoundRect metódus
TCanvas 132
Runtime Binding 37

S

Saját kivételek *lásd* Raise
SaveToFile metódus
TPicture 129
TStrings 81
Screen objektum 94
SDI alkalmazás 109
készítése 114
SelCount jellemző 72
SelectAll metódus 67
Selected jellemző 72
SelLength jellemző 66
SelStart jellemző 66
SelText jellemző 66
Sender paraméter 77
Server Manager 168
SetFocus metódus 65
ShortString típus 57
Show metódus 111
ShowMessage eljárás 106
ShowModal metódus 111
Sorted jellemző
TComboBox 74
Sovány (thin) kliens 162
SQL 269
DCL(Data Control Language) 269
DDL(Data Definition Language) 269
DML(Data Manipulation Language) 269
Jellemző
TQuery 270
Megadása a vizuális lekérdezésszerkesztővel 276
Megadása futási időben 278
Megadása tervezéskor a Database Desktop segítségével 274
Megadása tervezéskor begépeléssel 273
Standard komponensek 59
StartTransaction metódus
TDatabase 181
State

TDBDataset 182
State jellemző
TCheckBox 71
Statikus metódusok 37
Stretch jellemző
TImage 129
StretchDraw metódus
TCanvas 132
String típus 56
Strings jellemző 139
TStrings 80
StrToDate függvény 33
StrToDateTime függvény 33
StrToFloat függvény 33
StrToInt függvény 33
StrToTime függvény 33
Style jellemző
TComboBox 74
TFont 130
TTabset 102
Súgó készítése 353
Saját komponenshez 351
Származtatott mezők 205

T

TabIndex jellemző
TTabControl 100
TTabset 102
TableName jellemző
TTable 216
TabOrder 24
TabPosition jellemző
TPageControl 100
TTabControl 100
Tabs jellemző
TTabControl 100
TTabset 102
TabStop 24
TApplication 21
Tárolt eljárások 313, 317
TBitBtn komponens 68
TBitmap osztály 139
TBrush osztály 130
TButton komponens 68
TCanvas osztály 129
TCheckBox komponens 71
TComboBox 73
TComponent komponens 60
TControl komponens 62
TDatabase komponens 179
TDataModule osztály 171
TDataSource komponens 220
TDateTime típus 32
TDBComboBox komponens 229
TDBCtrlGrid komponens 226
TDBDataSet osztály 182
TDBGrid komponens 226
TDBListBox komponens 229
TDBLookupComboBox komponens 230
TDBLookupListBox komponens 230
TDBNavigator komponens 229
TDDEClientConv komponens 381
TDDEClientItem komponens 381
TEdit komponens 66
Telepítés 367
Text jellemző
TComboBox 74
TEdit 66
TextOut metódus
TCanvas 132
TField komponens 201
TFont osztály 130
TForm komponens 109
Thread
Tile metódus
TForm 111
TImage komponens 128
TimeToStr függvény 33
TLabel komponens 65
TList osztály 142
TListBox komponens 72
TMainMenu komponens 75
TMemo komponens 67
TMenuItem komponens 76
TMessage 40
TMessage típus 9, 347
TNoteBook komponens 101
TNotifyEvent típus 63
TObject osztály 34
TOLEContainer komponens 389

Top jellemző 63
Több rétegű (multi-tier) architektúra 162, 391
Több szálú alkalmazás 401
TPageControl komponens 100
TPen osztály 129
TPopupMenu komponens 76
TPrinter osztály 148
TQuery komponens 270
  Használata 271
TRadioGroup komponens 71
TransIsolation jellemző
  TDatabase 180
TRichEdit komponens 67
Triggerek 313
Try...Except blokk 50
Try...Finally blokk 53
TScreen osztály 94
TScrollCode 345
TScrollEvent 345
TSession
  Több szálú alkalmazásban 409
TSession komponens 178
TShape komponens 127
TSpeedButton komponens 70
TStrings osztály 80
TTabbedNotebook komponens 101
TTabControl komponens 100
TTable komponens 216
TTabset komponens 101
TThread osztály 402
TTimer komponens 85
TWinControl komponens 65

U

UML
  Láthatóság jelölése 41
Update metódus 65

Ü

Üzenet
  Fogalma 9
  Szétszórása és feldolgozása 12
  Típusai 11
  Üzenetlánc 12
Üzenetek
  Nem sorolt üzenetek 16
Üzenetkezelő ciklus 12
Üzenetsor
  Alkalmazás (szál) üzenetsora 12
  Windows üzenetsor 12

V

Vágólap 4, 379
Value jellemző
  TField 210
Variant 32
VCL 331
Védett 41
View *lásd* nézetek
Virtual key code *lásd* virtuális billentyű kód
Virtuális billentyű kód 10
Virtuális metódusok 37
Visible jellemző 63
  TField 212
Vizuális űrlapörökítés 114
VMT 37
Vonszolás technika 91

W

WideString típus 57
Width jellemző 63
  TPen 130
  TScreen 95
Window procedure
  lásd Ablakfüggvény 13
WindowMenu jellemző
  TForm 110
Windows
  Eseménykezelés, üzetnetvezérlés 9
  Memóriakezelés 4
Windows ISQL 168
WindowState jellemző

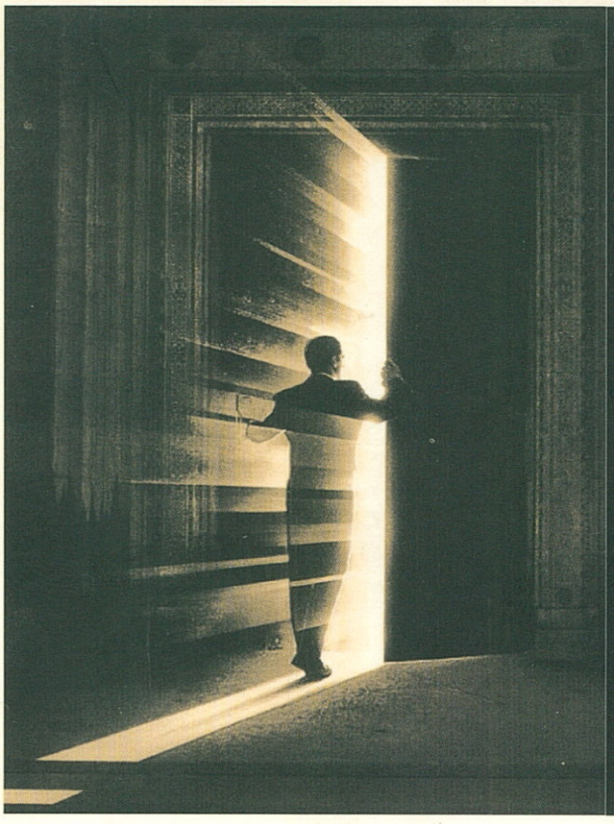
TForm 110
WM_ACTIVATE 16
WM_DESTROY 16
WM_KEYDOWN 10
WM_KEYUP 10
WM_LBUTTONDOWN 10
WM_QUIT 16
WM_SIZE 40
WM_VSCROLL 346

X

XOR és NotXOR rajzmódok 136

2.500, - Ft

A könyvben tárgyalt feladatok forráskódjai megtalálhatók az Interneten a www.borland.hu könyvek oldalon, vagy az ftp.gdf.hu/public/prog/delphi.zip állományban.

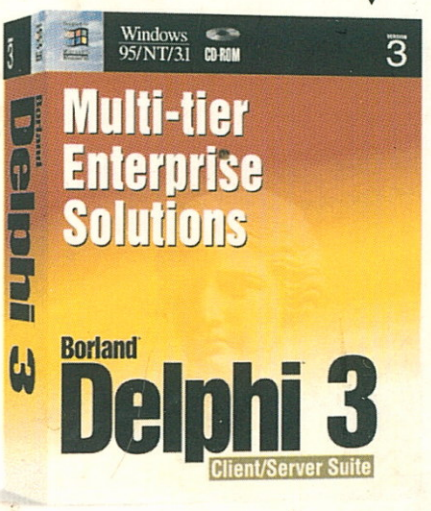


Delphi 3.0

Időben, költségvetésen belül!

Delphi 3.0 a legnagyobb teljesítményű RAD fejlesztőeszköz

A Delphit világszerte több mint hatszázezren használják, többek között olyan cégek, mint a BMW Finance, a NASA, az AT&T vagy az NEC. Miért teszik?



- A leggyorsabb és legtermelékenyebb fejlesztői környezet
- Többrétegű elosztott architektúra a kritikus alkalmazások támogatásához
- Egyszerű és hatékony internetes eszközök
- Nagy sebességű kapcsolat az összes fontos adatbázishoz, mint például: InterBase SQL, DB2, Oracle, MS SQL, MS Access, FoxPro dBASE, Paradox
- A Delphi for AS/400 segítségével a már megszokott eszközzel fejleszthet AS/400-as környezetben, nagyságrendekkel rövidítve a fejlesztési időt és csökkentve a fejlesztési költségeket
- A Delphi Connect for SAP az SAP teljes funkciókészletét elérhetővé teszi a Delphi programozók számára is, egyszerű kapcsolatot biztosítva ezáltal a PC-s világ és az SAP között.

SilVert

Borland Magyarország an Inprise Company

Borland Magyarország, 1143 Budapest, Hungária krt. 79-81., telefon: 252-8145
Fax: 252-8773, internet: http://www.borland.hu, e-mail: info@borland.hu